Theoretical Computer Science 30 (1984) 139–204
North-Holland

A CONSTRUCTIVE METHOD FOR ABSTRACT ALGEBRAIC SOFTWARE SPECIFICATION

Herbert A. KLAEREN

Lehrstuhl für Informatik II, RWTH Aachen, D-5100 Aachen, Fed. Rep. Germany

Communicated by M. Nivat
Received July 1982
Revised April 1983

Abstract. A constructive method for abstract algebraic software specification is presented, where the operations are not implicitly specified by equations but by an explicit recursion on the generating operations of an algebra characterizing the underlying data structure. The underlying algebra itself may be equationally specified since we cannot assume that all data structures will correspond to free algebras. This implies that we distinguish between generating and defined operations and that the underlying algebra has a mechanism of well-founded decomposition w.r.t. the generating operations. We show that the explicit specification of operations using so-called structural recursive schemata offers advantages over purely equational specifications, especially concerning the safeness of enrichments, the ease of semantics description and the separation between the underlying data structure and the operations defined on it.

Key words. Algebraic software specification, algebraic axioms, canonical term algebras, recursion schemata.

CR categories. D.3.3, F.3.1, F.3.2, D.2.1.

Contents

Introduction

When we consider software modules, it is obvious that they consist of two main components: the data part and the operation part. An abstract software specification should therefore also consist of a specification of some abstract data types (or data structures) and an abstract specification of operations. According to modern programming methodology, an abstract data type is not a collection of mere sets but of sets together with distinguished basic operations.

An abstract data type specification will therefore, in principle, be of the form $\langle S, \Sigma, \Delta \rangle$, where S is a set of *sorts* or *types* (which are names for carrier sets), Σ a set of *operations symbols* over S and Δ a set of *definitions* for Σ. The various approaches to abstract software specification differ essentially only in Δ. Objects like $\langle S, \Sigma \rangle$ happen to be familiar to mathematicians: they are many-sorted *signatures* determining categories $Alg_{\langle S,\Sigma \rangle}$ of S-sorted Σ-algebras. It is therefore legitimate to consider abstract data type specifications as presentations for algebras and to apply well-known results of universal algebra to data type specification problems.

From the algebraic point of view the most obvious approach is to take for Δ a set E of universally quantified equations between Σ-terms. $\mathbf{spec} = \langle S, \Sigma, E \rangle$ determines then a category $Alg_{\mathbf{spec}}$ which is also called an *algebraic variety* and which has always an initial algebra $T_{\mathbf{spec}}$, the abstract data type identified by **spec**.

Algebraic varieties have convenient mathematical properties; their use for abstract data type specification was first proposed by Zilles [63], the so called ADJ group (Goguen, Thatcher, Wagner, Wright: [28]) and Guttag [30]. Later developments allowed Δ to contain conditional axioms [29, 60], universal Horn sentences [19], existential quantifiers [8] and even inequalities and arbitrary first-order sentences [5–7]. (These latter approaches, of course, are no more algebraic but model-theoretic.)

There are two fundamental questions to be asked about specification methods: the question of *adequacy* and the question of *reliability*. It was first pointed out by Majster [49, 50] that the adequacy of equational specifications can be doubted, since certain simple data types are not finitely specifiable without so-called 'hidden functions'. Kamin [36] gave a formal framework for the *adequacy* question and Bergstra and Tucker [1, 3] showed that with a limited number of 'hidden functions', every computable data type is finitely specifiable.

Somewhat more connected to the practice of specifying abstract data types is the question of *reliability*: can we feel confident that a careful specification really expresses what we had in mind? It must be admitted that here the axiomatic specification method shows some deficiencies. If an algebraic specification is developed step by step in a systematic way then it seems very likely that one wants to add a new operation ('enrich'), possibly together with a new sort ('extend'), to an already existing specification **spec**. Of course the new operation is defined by further axioms, but the factorization induced by these new axioms can have unpleas-

ant consequences on previously defined operations or the set of possible data. At worst, T_{spec} breaks down to a one-element algebra.

The question of adequacy that was quoted above in a purely semantic context can also be asked with a pragmatic aspect: does the axiomatic technique allow to specify abstract data types such as they are, without artificial additions, unrealistic constraints or other peculiarities? Some cautious doubts seem to be well suited. It occurs, for instance, rather frequently that elements of data types have an observable *structure* which is reflected by structural operations such as *constructors* which generate new elements (possibly using old ones), *selectors* which give access to components of structured objects and *indicators* which identify the last constructor which has contributed to the generation of a specific data object. It is a rule that data types which do have constructors are inductively generated by these; all other operations can then be defined by their effect on terms build up with constructors. However, with the axiomatic approach all operations have equal rights; it is not possible to distinguish a priori *generating* and *defined* operations. In fact, this is one reason for the problems with enrichments or extensions mentioned before. It is, on the other hand, quite usual to insist on a *generation principle* for data types as described above: first we can prove theorems by induction and then we can define operations by a recursion on constructor terms.

Distinguishing between generating and defined operations, a data type becomes a special case of a software module consisting of a *structural component* describing the underlying data and a *functional component* defining the admissible operations of the data type. This author thinks that we should keep these two aspects as separated: constructors should be the first, basic thing if we design a data type; only afterwards we should think of defined operations. For recursively defined operations, we prefer to make the recursive definition explicit rather than coding it into some set of equations. Our specifications therefore are of the form $(S, \Sigma, E : \mathcal{C})$, where Σ is now the set of constructors, E (if non-empty) describes their properties (such as commutativity, associativity and the like) and possible relations between several constructors (such as one being inverse to the other). $\mathcal{C}$ is a set of certain *recursion schemata* for the definition of operations. These schemata are *constructive definitions* because they suggest a deterministic method for actually computing the application of the operations to some arguments, which is not true of equations. Loeckx [46, 47] and Nourani [53] also vigorously argue for constructive operation definitions. Our specifications have a lot in common with Loeckx's specifications: we need not care about consistency and completeness [30, 31] of the operation definitions; in particular, the enrichment/extension problem does not exist in either of these two approaches. (Section 10 contains a comparison of these two approaches.)

Recursive definitions cause no problem if the structural component of the data type corresponds to a term algebra, but this is not always the case. In the other cases the useful properties of the term algebra—structural induction and recursion—are recovered by certain *well-founded decomposition mappings* [11] which indicate how an element of an algebra can be generated by the constructors. Decomposition

mappings can be considered as a combination of the indicators and selectors mentioned before.

Note that we do not question the usefulness of equational or axiomatic specifications: they are quite valuable in the early stages of a specification development process when we only have some ideas about operation properties but are yet unable to describe them constructively. However, when a specification is more and more refined and developed, we must keep in mind that we probably aim at an implementation where we must of course eventually give algorithms. So axiomatic definitions have to be replaced step by step, as our insight into the problem grows, by constructive specifications. The present paper therefore not only studies the theory of constructive specifications but also gives a formal environment for proving that a constructive specification implements an axiomatic one.

1. Decomposition algebras

We must assume that the reader is familiar with the basic concepts of (many-sorted) universal algebra. Otherwise, introductions to this field can be found, e.g., in [27–29, 43]. We start this section by briefly fixing some notations.

Let ω denote the set of nonnegative integers; for $n \in \omega$ define $[n] := \{0, 1, \ldots, n\}$, especially $[0] := \emptyset$. For sets A, B let B^A denote the sets of total mappings, $[A \to B]$ the set of partial mappings from A to B. If S is a set, then an S-indexed family of sets, $A = (A^s)_{s \in S}$, is called an *S-set*. The elements of S are also called *sorts*, and A an *S-sorted set*. For a finite word $w \in S^*$, $w = s_1 \ldots s_n$, define $A^w := A^{s_1} \times \cdots \times A^{s_n}$. A^ε is a singleton set, where $\varepsilon \in S^*$ is the empty word; we denote the single element of A^ε by $( )$. If A and B are S-sets, a mapping $f: A \to B$ is called *sort-preserving* (or an S-mapping) if it can be represented as a family $f = (f^s: A^s \to B^s)_{s \in S}$.

Any mathematical construction between S-sets in the present paper should be understood as its sort-preserving analogue, unless the contrary is stated. For a set S of sorts, the elements of $D(S) := S^* \times S$ are called the *derived sorts of S*. A pair $\mathbf{sig} = (S, \Sigma)$ is called a *signature* if Σ is a $D(S)$-set. $\Sigma^{(w,s)}$ is called the set of *operation symbols* in Σ with arity w and target sort s. Let $\Sigma^{(s)} := \bigcup_{w \in S^*} \Sigma^{(w,s)}$. For an S-set A, $w \in S^*$ and $s \in S$, define $\mathrm{Ops}^{(w,s)}(A) := \{f: A^w \to A^s\}$, the set of *operations* on A. Special operations are the *projections* $\pi_i^w: A^w \to A^{s_i}$ for $w = s_1 \ldots s_n \in S^*$ and $i \in [n]$, with $\pi_i^w(a_1, \ldots, a_n) = a_i$. We also allow a degenerate projection $\pi_0^w: A^w \to A^\varepsilon$ with $\pi_0^w(a_1, \ldots, a_n) = ( )$.

If $\mathbf{sig} = (S, \Sigma)$ is a signature, then a $\mathbf{sig}$-algebra A is an S-set A together with operations $F_A \in \mathrm{Ops}^{(w,s)}(A)$ for every $F \in \Sigma^{(w,s)}$. If S is obvious, we also speak of a Σ-algebra. If Σ is finite, we also write $A = (A; F_1^{(w_1,s_1)}, \ldots, F_n^{(w_n,s_n)})$.

We assume that the reader knows the notions of homomorphism, subalgebra, reduct, generating set, free algebra and so on. The subalgebra of A generated by $X \subseteq A$ is denoted by $\hat{X}$, where $\hat{X}$ is the closure of X under the operations of Σ. The following facts are well-known from universal algebra:

The class of all **sig**-algebras together with the class of all homomorphisms forms a category $Alg_{\mathbf{sig}}$.

Lemma 1.1 (Unique extension). *Let A be a* **sig***-algebra generated by X, B a* **sig***-algebra and $f: X \to B$ a sort-preserving mapping. Then there is at most one homomorphism $h: A \to B$ with $h(x) = f(x)$ for all $x \in X$.*

Theorem 1.2. *In every class $\mathcal{K}$ of* **sig***-algebras there is up to isomorphism at most one free* **sig***-algebra with free generating set X.*

Definition (Term algebra). Let $\mathbf{sig} = \langle S, \Sigma \rangle$ be a signature, X an S-set. Define the set $T_{\mathbf{sig}}(X)$ of **sig**-terms on X as the smallest S-subset $T \subset (\Sigma \cup X)^*$ with the properties

(i) $(\forall s \in S) \quad X^s \subseteq T^s$,

(ii) $(\forall s \in S) \quad \Sigma^{(\varepsilon, s)} \subseteq T^s$,

(iii) $(\forall F \in \Sigma^{(w,s)})(\forall t \in T^w) \quad Ft \in T^s$.

$T_{\mathbf{sig}}(X)$ is turned into a **sig**-algebra by

(iv) $(\forall F \subset \Sigma^{(\varepsilon, s)}) \qquad F_{T_{\mathbf{sig}}}(\ ) := F$,

(v) $(\forall F \in \Sigma^{(w,s)})(\forall t \in T_{\mathbf{sig}}(X)^w) \quad F_{T_{\mathbf{sig}}}(t) := Ft$.

$T_{\mathbf{sig}}(X)$ is called the **sig**-term algebra on X. Because of (iv), (v) we usually write F instead of $F_{T_{\mathbf{sig}}}$.

Theorem 1.3 (Unique term decomposition). *For every $t \in T_{\mathbf{sig}}(X)$ either $t \in X$ or there are unique $F \in \Sigma, t_1, \ldots, t_n \in T_{\mathbf{sig}}(X)$ such that $t = Ft_1 \ldots t_n$.*

Theorem 1.4. *$T_{\mathbf{sig}}(X)$ is the free* **sig***-algebra in $Alg_{\mathbf{sig}}$ with free generating set X.*

(This is a direct consequence of unique term decomposition.)

Theorem 1.5. *For every* **sig***-algebra A and every generating set X of A the unique homomorphism $h_A: T_{\mathbf{sig}}(X) \to A$ given as the homomorphic extension of the identity $i: X \to A$ is surjective.*

A stronger version of this theorem can be obtained using *congruence relations*. Define a relation $\equiv_A$ on $T_{\mathbf{sig}}(X)$ by $t_1 \equiv_A t_2 :\Leftrightarrow h_A(t_1) = h_A(t_2)$. It is easily seen that this is a congruence relation, and that the following holds.

Theorem 1.6. *For every generating set X of a* **sig***-algebra A, A is isomorphic to $T_{\mathbf{sig}}(X)/\equiv_A$.*

This means that every **sig**-algebra can be considered as a factor algebra of some free algebra. Especially important is the free algebra on no generators.

Definition. The free algebra in Alg_{sig} with generating set $\emptyset$ is called the *initial* **sig**-algebra. It is denoted by T_{sig} (instead of $T_{sig}(\emptyset)$).

Corollary 1.7. *For every* **sig***-algebra* A *there is a unique homomorphism* $h_A : T_{sig} \to A$. *If* A *is generated by* $\emptyset$, *then* h_A *is surjective.*

For the remainder of this paper, we will usually not consider $T_{sig}(X)$ for $X \neq \emptyset$ nor other **sig**-algebras not generated by the empty set. This means that we restrict ourselves to *standard models* for **sig**-theories: if **sig**-algebras are considered as data types and the term algebra T_{sig} as an abstract description of the data types (cf. [43]) then all terms in T_{sig} denote a unique element of each **sig**-algebra A via the unique homomorphism $h_A : T_{sig} \to A$. Now if A is not generated by $\emptyset$, then h_A is not surjective and there are elements of A which cannot be denoted by an 'abstract name'. Since we start from the principle that the essential features of any data type can be derived from its abstract presentation, it seems best to ignore elements which are not in the range of h_A. $T_{sig}(X)$ for $X \neq \emptyset$ is not an appropriate domain for abstract names though it can be made rich enough to cover any **sig**-algebra (cf. Theorem 1.6): a term $t \in T_{sig}$ denotes an element $h_A(t)$ *automatically*, i.e., without further explanation, whereas $t' \in T_{sig}(X)$ for $X \neq \emptyset$ can only be evaluated by $\hat{\theta}(t')$ for some assignment $\theta : X \to A$ of variables. Different assignments generally lead to different elements of A, so there is no unique object in A corresponding to t'.

As we said before, every **sig**-algebra is isomorphic to a factor algebra of some free algebra. Of course, there is special interest in the case where this factorization can be effectively described. From the algebraic point of view it seems most natural to use sets of *algebraic equations*. Depending on what kind of equation sets are admitted, different classes of factorizations and data types are specifiable, see [1, 3, 36]. In the following paragraphs we briefly review some basic facts about equations; details can be found in [29] or any textbook on universal algebra. Equations can also be considered as rewrite rules for some subtree (= subterm) replacement system; we will not concentrate on this interpretation of equations but instead refer to the concise review of Huet and Oppen [34].

Let X be an S-sorted standard alphabet with $X^{\omega} = \{x_i^{(s)} \mid i < \omega\}$.

Definition 1.8. An S-mapping $\theta : X \to A$ for a **sig**-algebra A is called an *interpretation* (of X). A relation $E \subseteq T_{sig}(X)^2$ is called a set of **sig**-equations. A set of **sig**-equations induces by the interpretations of X a relation E_A for every **sig**-algebra A:

$$E_A := \{(\hat{\theta}(t_1), \hat{\theta}(t_2)) \mid (t_1, t_2) \in E, \theta : X \to A \text{ interpretation}\}.$$

Here $\hat{\theta}$ denotes the homomorphic extension of θ.

A class $\mathcal{K}$ of **sig**-algebras *satisfies* E iff

$$(\forall A \in \mathcal{K})(\forall (t_1, t_2) \in E)(\forall \theta : X \to A) \quad \hat{\theta}(t_1) = \hat{\theta}(t_2).$$

E is then also called *valid in* $\mathcal{K}$.

Usually, we write equations as $t_1 = t_2$ instead of (t_1, t_2). In Definition 1.8 it suffices to consider all interpretations $\vartheta : Y \to A$, where Y is the set of variables occurring in t_1 and t_2. This is always a finite set.

Definition. $Alg_{(\mathbf{sig}, E)}$ denotes the subcategory of $Alg_{\mathbf{sig}}$ in which E is valid.

$Alg_{(\mathbf{sig}, E)}$ is determined or specified by the triple $\langle S, \Sigma, E \rangle$ which we therefore also call an (*equational*) *specification.*

Theorem 1.9. *Let* **spec** $= \langle S, \Sigma, E \rangle$ *be a specification,* $\equiv_E$ *the congruence generated by* $E_{T_{\mathbf{sig}}}$ *on* $T_{\mathbf{sig}}$. *Then* $T_{\mathbf{spec}} := T_{\mathbf{sig}} / \equiv_E$ *is the initial algebra in* $Alg_{\mathbf{spec}}$.

Definition. If **spec** $= \langle$**sig**, $E \rangle$ is a specification, then $\text{SYN}_{\mathbf{spec}} := T_{\mathbf{sig}}$ is called the *syntactic algebra* of **spec** and $\text{SEM}_{\mathbf{spec}} := T_{\mathbf{spec}}$ is called the *semantic algebra* of **spec**.

The unique homomorphism $h : \text{SYN}_{\mathbf{spec}} \to \text{SEM}_{\mathbf{spec}}$ is called the (*initial*) *semantics of* **spec**.

Elements of $\text{SEM}_{\mathbf{spec}}$ are congruence classes $[t]$ of terms $t \in \text{SYN}_{\mathbf{spec}}$. If we want to denote these classes, we must of course choose a *system of representatives*. In general this system will not be closed under term decomposition, i.e., if $Ft_1 \ldots t_n$ is a representative in the system, then the respective representatives of $[t_1], \ldots, [t_n]$ are not necessarily $t_1, \ldots, t_n$. But there always exists a systematic or 'canonical' system of representatives for $\text{SEM}_{\mathbf{spec}}$.

Definition 1.10 ([29]). A **sig**-algebra C is called a canonical **sig**-term algebra (**sig**-CTA, for short): $\Leftrightarrow$

(i) $C_s \subseteq T_{\mathbf{sig}, s}$ for all $s \in S$,

(iia) $Ft_1 \ldots t_n \in C \Rightarrow (\forall i \in [n]) \quad t_i \in C$,

(iib) $Ft_1 \ldots t_n \in C \Rightarrow F_C(t_1, \ldots, t_n) = Ft_1 \ldots t_n$.

Next we want to show that for every **sig**-algebra A generated by $\emptyset$ there is a **sig**-CTA $C \cong A$. (This has been proved in [29] but we want to give an alternative proof which gives some hints on the algorithmic construction of C in those cases where this can be accomplished.) For this proof, we need a definition and an auxiliary lemma (see [59]).

Definition. A partial ordering $\leq$ on $T_{\mathbf{sig}}$ is called *substitution closed* iff, for all $Ft_1 \ldots t_n, Ft'_1 \ldots t'_n \in T_{\mathbf{sig}}$ we have $Ft_1 \ldots t_n \leq Ft'_1 \ldots t'_n$ whenever $t_i \leq t'_i$ for all $i \in [n]$.

Lemma 1.11. *Let $\leq$ be a substitution closed partial ordering on $T_{\mathbf{sig}}$ and let A be a* **sig***-algebra generated by $\emptyset$. If the set $[t] := \{t' \in T_{\mathbf{sig}} \mid t \equiv_A t'\}$ has for all $t \in T_{\mathbf{sig}}$ an infimum w.r.t. $\leq$, then there is a* **sig**-CTA $C \cong A$.

Proof. Define $\rho : T_{\mathbf{sig}} \to T_{\mathbf{sig}}$ by $\rho(t) := \inf([t])$, $C := \rho(T_{\mathbf{sig}})$ and $F_C(t_1, \ldots, t_n) := \rho(Ft_1 \ldots t_n)$. Then it is easy to show that C is isomorphic to A. For the proof that C is decomposition closed (Definition 1.10(iia)) assume $Ft_1 \ldots t_n \in C$ and $t_j \notin C$ for some $j \in [n]$. Then $t' := \rho(t_j) \in C$, $t' \leq t_j$, and $t' \equiv_A t_j$. This implies $Ft_1 \ldots t' \ldots t_n \equiv_A Ft_1 \ldots t_j \ldots t_n$, hence $Ft_1 \ldots t' \ldots t_n \in [Ft_1 \ldots t_j \ldots t_n]$. But $\leq$ is substitution closed, so $Ft_1 \ldots t' \ldots t_n \leq Ft_1 \ldots t_j \ldots t_n$ which means that $Ft_1 \ldots t_j \ldots t_n$ cannot be the representative of its congruence class in C unless $t' = t_j$. This, however, contradicts the initial assumption. The proof for Definition 1.10(iib) is straightforward. □

Theorem 1.12. *Let A be a* **sig***-algebra generated by $\emptyset$. Then there is a* **sig**-CTA C *isomorphic to A.*

Proof. Let $\leq$ be any well-founded total ordering of Σ. $\leq$ is extended to a well-founded ordering on Σ^* by

$$F_1 \ldots F_n \leq F'_1 \ldots F'_m \text{ iff either (i) } n < m, \text{ or}$$
$$\text{(ii) } n = m \text{ and there is } j \in [n] \text{ with } F_j \leq F'_j \text{ and } F_i = F'_i \text{ for all } i \in [j-1].$$

Then $\leq$ is substitution closed and every nonempty subset of $T_{\mathbf{sig}}$ has an infimum w.r.t. $\leq$. So by Lemma 1.11, there is a **sig**-CTA $C \cong A$. □

In general, it is not decidable whether $t \equiv_A t'$ for $t, t' \in T_{\mathbf{sig}}$; this is the so-called *word problem* for A. If, however, the word problem for a special algebra A is decidable, then Theorem 1.12 together with Lemma 1.11 gives a method for the construction of C: In order to find the canonical representative $\rho(t)$ for some $t \in T_{\mathbf{sig}}$, we compare with all t' in ascending order w.r.t. $\leq$ until we find the smallest t' with $t \equiv_A t'$. (Note that $T_{\mathbf{sig}}$ is totally ordered by $\leq$.)

Definition ([3]). A **sig**-algebra N is called a *recursive* **sig***-number algebra* iff for all $s \in S$ $N_s \subseteq \omega$ and for all $F \in \Sigma$ F_N is a partial recursive function. A **sig**-algebra A is a *computable* algebra iff there is a recursive **sig**-number algebra N and an epimorphism $h : N \to A$ such that

$$\equiv_h := \{(x, y) \in \omega^2 \mid h(x) = h(y)\}$$

is decidable.

Theorem 1.13 (Schulz [59]). *Let A be a* **sig***-algebra generated by $\emptyset$ with finite Σ. Then the following statements are equivalent:*

(i) ***A is computable.***

(ii) ***The word problem for A is decidable.***

(iii) ***There is a canonical term algebra C isomorphic to A such that every F_C for $F \in \Sigma$ is a partial recursive word function.***

Proof. For the proof, see [44, 59]. □

Note that a canonical term algebra—though being contained in $T_{\mathbf{sig}}$—will generally not be a subalgebra of $T_{\mathbf{sig}}$.

Term algebras have pleasing mathematical properties: the existence of homomorphic extensions guarantees the unique existence of certain recursively defined operations and we can prove assertions by induction on the structure of terms. Both of these properties get lost if we pass to non-free algebras. In the following paragraphs we will show that they can be partially recovered if we 'simulate' the effect of unique term decomposition.

Definition. Let A be a **sig**-algebra generated by X. A mapping $d: A \to X \cup (\Sigma \times A^*)$ is called a *decomposition of* A w.r.t. $X :\Leftrightarrow$ for all $a \in A$,

$$d(a) = (F, b_1 \ldots b_k) \Rightarrow F_A(b_1, \ldots, b_k) = a \quad \text{(including the case } k = 0),$$

$$d(a) = x \in X \Rightarrow a = x.$$

Define $a \sqsubset_d b$ iff $d(b) = (F, c_1 \ldots c_k)$ and $a = c_i$ for some $i \in [k]$. Then d is called *well-founded* iff there is no infinite descending chain

$$a_0 \sqsupset_d a_1 \sqsupset_d a_2 \sqsupset_d \cdots.$$

Let $<_d$ denote the transitive closure of $\sqsubset_d$.

It is trivial that, for a well-founded decomposition, $a <_d b$ implies $a \neq b$; in particular, for $d(b) = (F, c_1 \ldots c_k)$ we have $(\forall i \in [k])\ c_i \neq b$.

Example 1.14. Consider the one-sorted algebra $\boldsymbol{\omega}/\equiv_{\mathrm{mod}(p+1)}$ which is given by $([p] \cup \{0\}; 0^{(0)}, suc^{(1)})$ with the following interpretation of operation symbols:

$$0(\,) := 0, \qquad suc(n) := \begin{cases} n+1 & \text{if } n < p, \\ 0 & \text{if } n = p. \end{cases}$$

There are two different decompositions of $\boldsymbol{\omega}/\equiv_{\mathrm{mod}(p+1)}$ (w.r.t. $\emptyset$):

$$d(0) = (0, (\,)), \qquad d'(0) = (suc, p).$$

$$d(n) = (suc, n-1), \qquad d'(n) = (suc, n-1), \quad \text{for } n \in [p].$$

d is well-founded, d' not.

An alternative characterization of $\omega/\equiv_{\mathrm{mod}(p-1)}$ can be given by a specification $\mathbf{spec} = \langle S, \Sigma, E\rangle$ with

$$S = \{s\}, \quad \Sigma = \{0^{(\epsilon,s)}, suc^{(s,s)}\} \quad \text{and} \quad E = \{suc^{p+1}(0) = 0\}.$$

Then there is a unique canonical **sig**-algebra $C \cong \mathrm{SEM}_{\mathbf{spec}}$, and $\omega/\equiv_{\mathrm{mod}(p+1)}$ is isomorphic to C by $n \leftrightarrow suc^n(0)$.

The well-founded decomposition d mentioned above has a special importance for the algebra C: we call it the decomposition *inherited* from the syntactic algebra $\mathrm{SYN}_{\mathbf{spec}}$.

Decomposition mappings were first introduced in a similar context by Burstall and Landin [11] for the definition of homomorphic extensions in non-free algebras. Frequently, a decomposition is split up into an *indicator* which identifies the operation by which a certain element is generated and a set of *selectors* which select the respective arguments of this operation (see the comments in the Introduction).

Definition 1.15. An *X-well-founded* **sig**-*algebra* is a **sig**-algebra A generated by $X \subseteq A$ together with a well-founded decomposition d_A of A w.r.t. X. A **sig**-*decomposition algebra* $\boldsymbol{A}$ (**sig**-d-algebra, for short) is a d_A-well-founded **sig**-algebra A.

Speaking of $\boldsymbol{A}$ as a **sig**-d-algebra always will imply that the primitive operations are denoted by F_A and the well-founded decomposition by d_A.

Now we show two important properties of X-well-founded **sig**-algebras.

Theorem 1.16 (Structural induction). *Let A be a X-well-founded* **sig**-*algebra, P a predicate on A. Then*

$$[(\forall a \in A)\,[(\forall b <_{d_A} a)\; Pb] \Rightarrow Pa] \Rightarrow (\forall a \in A)\; Pa.$$

Proof. Let P be a total predicate on A with

$$[(\forall a \in A)[(\forall b <_{d_A} a) Pb] \Rightarrow Pa]. \qquad (*)$$

Let $a_0 \in A$ such that Pa_0 is false. According to (*) there must exist $a_1 \in A$ such that $a_1 <_{d_A} a_0$ and Pa_1 is false. Note that $a_1 \neq a_0$. This argument can be repeated arbitrarily giving an infinite descending chain $a_0 >_{d_A} a_1 >_{d_A} a_2 >_{d_A} \cdots$, in contradiction to the well-foundedness of d_A. □

The elements of X are minimal w.r.t. $<_{d_A}$. So if we want to prove some fact about an X-well-founded **sig**-algebra, we only have to show

(1) that the assertion is true for the generating set X (if non-empty), and

(2) that (*) in Theorem 1.16 is valid.

Theorem 1.17 (Recursion Theorem). *Let $\boldsymbol{A}$ be an X-well-founded* **sig**-*algebra, B a* **sig**-*algebra, $f: X \to B$ a sort-preserving mapping. Then there is a unique mapping*

$h_f : A \to B$ with the property

$$(\forall a \in A) \quad h_f(a) = \begin{cases} f(a) & \text{for } a \in X, \\ F_B(h_f^w(\boldsymbol{c})) & \text{for those } F \in \Sigma,\ \boldsymbol{c} \in A^w \text{ with} \\ & d_A(a) = (F, \boldsymbol{c}). \end{cases}$$

If f can be extended to a homomorphism $\hat{f} : A \to B$, then $\hat{f} = h_f$.

Proof. (A) Existence of h_f. Define a relation g by

$$g := \bigcap \{T \subseteq A \times B \mid (a, f(a)) \in T \text{ for all } x \in X \text{ and if } (a_1, b_1), \ldots, (a_r, b_r) \in T \text{ and } d_A(a) = (F, (a_1, \ldots, a_r)), \text{ then } (a, F_B(b_1, \ldots, b_r)) \in T\}.$$

By structural induction we show that g is indeed the graph of a function h_f which trivially has the required property.

So we have to show $(\forall a \in A)(\exists^1 b \in B)\ (a, b) \in g$.

For all $x \in X$ this is clear since g is the smallest graph containing $(a, f(a))$. Now let $a \in A \setminus X$ and assume that the assertion is true for all $y <_{d_A} a$. Let $d_A(a) = (F, (a_1, \ldots, a_r))$; then $a_i <_{d_A} a$ for all $i \in [r]$ and $a_i \neq a$ since d_A is well-founded. By induction hypothesis, we have $(\forall i \in [r])(\exists^1 b_i \in B)\ (a_i, b_i) \in g$. Now by definition of g we have $(\exists^1 b = F_B(b_1, \ldots, b_r))\ (a, b) \in g$.

(B) Uniqueness. Let $h : A \to B$ be a further mapping with the property of h_f. First it is clear that $(\forall a \in X)\ h_f(a) = f(a) = h'(a)$. Now let $a \in A$, $d_A(a) = (F, \boldsymbol{c})$ and $(\forall y <_{d_A} a)\ h_f(y) = h'(y)$. Then $h'(a) = F_B(h'^w(\boldsymbol{c})) = F_B(h_f^w(\boldsymbol{c})) = h_f(a)$.

(C) If $\phi : A \to B$ is the extension of f to a homomorphism, then it clearly has the property of h_f. By (B) this implies $\phi = h_f$. □

The Recursion Theorem 1.17 is a generalization of homomorphic extendability, and X-well-founded algebras have properties very close to those of free algebras. Generally, we will restrict our attention to **sig**-d-algebras; see the remarks following Corollary 1.7.

Lemma 1.18. *For every* **sig***-algebra A generated by $\emptyset$ there is a well-founded decomposition d_A of A* w.r.t. $\emptyset$.

Proof. First, by Theorem 1.12, there is a **sig**-CTA $C \cong A$. Let $d_{\mathbf{sig}}$ denote the decomposition of C inherited from $T_{\mathbf{sig}}$ and i denote the isomorphism $i : C \to A$. Now define, for $a \in A$,

$$d_A(a) := (F, i^w(\boldsymbol{t})), \quad \text{where } a = i(c),\ c \in C, \quad d_{\mathbf{sig}}(c) = (F, \boldsymbol{t}),\ F \in \Sigma^{(w,s)},\ \boldsymbol{t} \in C^w.$$

It remains to show that d_A is indeed a decomposition of A. If $d_A(a) = (F, i^w(\boldsymbol{t}))$, then $a = i(c) = i(F\boldsymbol{t})$. This implies $F_A(i^w(\boldsymbol{t})) = i(F\boldsymbol{t}) = a$. It is easily shown by induction that d_A is well-founded. □

It is therefore no essential restriction to consider only **sig**-d-algebras instead of arbitrary finitely generated **sig**-algebras. However, two comments must be given: First, the proof of Lemma 1.18 is non-constructive in the general case since it involves the non-constructive Theorem 1.12. Second, there may be several ways to turn a **sig**-algebra into a **sig**-d-algebra, so the transition from a **sig**-algebra to a corresponding **sig**-d-algebra is not a unique one.

Corollary 1.19. *Let A be a computable* **sig***-algebra generated by $\emptyset$. Then there is an algorithm to construct a computable well-founded decomposition d_A of A w.r.t. $\emptyset$.*

Proof. For the proof we investigate the constructions of Lemma 1.18. First, we have to construct a **sig**-CTA $C \cong A$. By Theorem 1.13, the word problem for A is decidable, so the remarks after Theorem 1.12 give an algorithm for the construction of C. By Theorem 1.13, every F_C for $F \in \Sigma$ is partial recursive; this implies that the unique isomorphism $i: C \to A$ is also partial recursive. Obviously, this also is true for i^{-1} and clearly $d_{\mathbf{sig}}$ is computable. It follows that d_A, as constructed in Lemma 1.18, is computable. □

If the ordering $\leqslant$ on Σ used in the proof of Theorem 1.12 is fixed, then this procedure yields a unique computable well-founded decomposition for every finitely generated computable **sig**-algebra.

The results obtained so far suggest that there are very close relationships between **sig**-CTA's and **sig**-d-algebras: First, every **sig**-CTA is, by definition, also a **sig**-d-algebra, and second, **sig**-CTA's play an essential role in the construction of **sig**-d-algebras. We will now show that a **sig**-d-algebra also implies a **sig**-CTA with special properties.

First we note that the algebras A and C in Lemma 1.18 are compatible in the following sense.

Definition. Let $\mathbf{A}_0, \mathbf{A}_1$ be **sig**-d-algebras with decompositions d_0, d_1. If $f: A_0 \to A_1$ is a mapping, then A_0 and A_1 are called *compatible* w.r.t. $f :\Leftrightarrow$ for all $b \in A_0$, $d_1(f(b)) = (F, a_1 \ldots a_k)$ implies $d_0(b) = (F, b_1 \ldots b_k)$ and $f(b_i) = a_i$ for all $i \in [k]$.

Lemma 1.20. *For every* **sig**-d-*algebra* $\mathbf{A}$ *there is a* **sig**-CTA C *and an isomorphism $i: C \to A$ such that A and C are compatible w.r.t. i.*

Proof. By Theorem 1.17, we can define a unique mapping $f: A \to T_{\mathbf{sig}}$ by $f(a) := Ff^w(\boldsymbol{b})$ for $d_A(a) = (F, \boldsymbol{b})$, $\boldsymbol{b} \in A^w$. f is injective, because $d_A(x) = d_A(y)$ implies $x = y$. Obviously, the unique homomorphism $h_A: T_{\mathbf{sig}} \to A$ satisfies

$$h_A \circ f = \mathrm{id}_A.$$

(However, f will in general not be a homomorphism.)

Now let $C := \{f(a) \mid a \in A\}$; then $f: A \to C$ and $h'_A: C \to A$ are bijections, where h'_A is the appropriate restriction of h_A.

For all $F \in \Sigma^{(w,s)}$, $t \in C^w$ define

$$F_C(t) := f(F_A(h_A^w(t))).$$

Then C is a **sig**-CTA, because we have

(i) $C \subseteq T_{\mathbf{sig}}$,

(iia) $Ft_1 \ldots t_n \in C \Rightarrow Ft_1 \ldots t_n = f(a)$ for some $a \in A$ with $d_A(a) = (F, b_1 \ldots b_n)$ and $t_i = f(b_i)$ for all $i \in [n]$. This means $t_i \in C$ for all $i \in [n]$.

(iib) $Ft_1 \ldots t_n \in C \Rightarrow F_C(t_1, \ldots, t_n) = f(F_A(h_A(t_1), \ldots, h_A(t_n)))$

$$= f(F_A(h_A(f(b_1)), \ldots, h_A(f(b_n)) \ldots))$$

$$= f(F_A(b_1, \ldots, b_n)) = f(a) = Ft_1 \ldots t_n.$$

Furthermore, we have $f \circ h'_A = \mathrm{id}_C$, because we have $c = f(a)$ for all $c \in C$ some $a \in A$ and

$$(f \circ h'_A)(c) = f(h'_A(f(a))) = f(a) = c.$$

Using this, we can show that $f: A \to C$ is a homomorphism: For all $F \in \Sigma^{(w,s)}$, $a \in A^w$:

$$f(F_A(a)) = f(F_A(h_A'^w(f^w(a))))$$

$$= f(h'_A(F_C(f^w(a)))) \quad \text{(because } h'_A \text{ is a homomorphism)}$$

$$= F_C(f^w(a)).$$

It is obvious that A and C are compatible w.r.t. f^{-1}. □

We will now show that in a **sig**-d-algebra the equality predicate is decidable, implying that a **sig**-d-algebra is always a computable algebra.

Definition 1.21. Let A be a **sig**-d-algebra, $a, b \in A$. Define

$$a \equiv_{d_A} b :\Leftrightarrow d_A(a) = (F, a_1 \ldots a_n),\ d_A(b) = (F, b_1 \ldots b_n) \text{ and}$$

$$a_i \equiv_{d_A} b_i \text{ for all } i \in [n].$$

It is obvious that $\equiv_{d_A}$ is an equivalence relation and, moreover, a decidable one. Also it is clear that for $f: A \to T_{\mathbf{sig}}$, as in the proof of Lemma 1.20, we have

$$a \equiv_{d_A} b \Leftrightarrow f(a) = f(b).$$

This can be shown, for instance, by term induction on $f(a)$. Using $h_A \circ f = \mathrm{id}_A$, we

see that $a \equiv_{d_A} b$ if and only if

$$a = h_A(f(a)) = h_A(f(b)) = b.$$

So we have proved the following.

Corollary 1.22. *Equality in a* **sig**-d-*algebra is decidable.*

2. Recursive definitions

Following the comments given in the Introduction, we will now investigate methods for defining new operations over an already defined algebra. Here, the algebra is thought of as describing an underlying data structure on which we want to specify higher functions. We can assume that the underlying algebra A is finitely specified, i.e., $A = \text{SEM}_{\textbf{spec}}$ for a finite $\textbf{spec} = \langle S, \Sigma, E \rangle$.

One possibility is, of course, to enlarge the operation alphabet Σ and to specify the new operation by further equations. This means that we pass to a different signature $\textbf{sig}' = \langle S, \Sigma' \rangle$, $\Sigma \subseteq \Sigma'$ and a different specification $\textbf{spec}' = \langle \textbf{sig}', E' \rangle$, $E \subseteq E'$. We will later comment on advantages and disadvantages of this procedure (Section 4). At this place, we will avoid equations as a defining mechanism.

If the specification for the underlying algebra does not involve equations, i.e., $\textbf{spec} = \langle S, \Sigma, \emptyset \rangle$, then $\text{SEM}_{\textbf{spec}} = \text{SYN}_{\textbf{spec}}$ is (isomorphic to) a term algebra. Hupbach [35] has studied the properties of certain recursively defined tuple-valued operations on term algebras, calling them *partial recursive tree functions*. After defining the functionals of primitive recursion and of iteration (instead of minimalization which is only applicable assuming a more or less artificial well-ordering of terms) he defines the class of primitive recursive tree functions as the closure of all projections and the basic operations of Σ under composition, target tupling and primitive recursion, the class of partial recursive tree functions as being in addition closed under iteration, and the *general recursive tree functions* as the total functions among the partial recursive ones. The main result of Hupbach's paper is that the i-recursive tree functions ($i \in \{\text{primitive, partial, general}\}$) over a signature $\textbf{sig} = \langle S, \Sigma \rangle$ are exactly those functions corresponding to i-recursive number-theoretic functions via a standard enumeration of $T_{\textbf{sig}}$, if $T^s_{\textbf{sig}}$ is infinite for at least one $s \in S$.

Unfortunately, not all underlying algebras we may want to consider in software specification applications correspond to a term algebra. We have therefore investigated the possibility of recursive definitions in non-term-algebras. (For an alternative to this approach we refer to Loeckx [46, 47]; see the comments in Section 10.) It is obvious that we cannot expect even primitive recursion to work on an arbitrary **sig**-algebra A since a kind of *unique decomposition* of the recursion argument is inherent in the primitive recursive definition. Things look different if we proceed to **sig**-d-algebras, as we will show in this section. Remember that a well-founded decomposition always exists and, in the case of a computable algebra, can even be constructed.

For the purpose of comparison we want to reformulate Hupbach's definition of primitive recursion in our notation.

Theorem 2.1 ([35, Satz 2.1]). *Let* $\mathbf{sig} = \langle S, \Sigma \rangle$ *be a signature,* $v \in S^*$ *and* $u_s \in S^*$ *for all* $s \in S$. *For* $w = w_1 \dots w_n$ *let* $u_w := u_{w_1} \dots u_{w_n}$. *Then for every* Σ*-indexed family* g *of functions such that*

$$g_F : T^v_{\mathbf{sig}} \times T^w_{\mathbf{sig}} \times T^{u_w}_{\mathbf{sig}} \to T^{u_s}_{\mathbf{sig}} \quad \text{for } F \in \Sigma^{(w,s)},\ w = w_1 \dots w_n \in S^*,$$

there is a unique S*-indexed family* h *of operations* $h = (h^s)_{s \in S}$, *where*

$$h^s : T^v_{\mathbf{sig}} \times T^s_{\mathbf{sig}} \to T^{u_s}_{\mathbf{sig}},$$

with the property that, for all $x \in T^v_{\mathbf{sig}}$, $s \in S$, $Ft_1 \dots t_n \in T^s_{\mathbf{sig}}$ *we have*

$$h^s(x, Ft_1 \dots t_n) = g_F(x, t_1, \dots, t_n, h^{w_1}(x, t_1), \dots, h^{w_n}(x, t_n)).$$

If the g_F *are partial, there is still a minimally defined family* h *which has the properties mentioned above.*

Definition. Under the premises of Theorem 2.1, $h = (h^s)_{s \in S}$ is said to be generated by $g = (g_F)_{F \in \Sigma}$ using primitive recursion, $h = \mathrm{PR}[g]$.

As we have already mentioned, this definition allows tuple-valued functions. If, for all $s \in S$, $\lg(u_s) = 1$, then we have the single-valued case of a primitive recursive definition which involves a simultaneous recursion on all sorts $s \in S$. Frequently, an operation h defined by recursion will only have one recursive argument and return a single value. In this case we have $u_s = t$ for some $s, t \in S$ and $u_x = \varepsilon$ for all $s \neq x$, $x \in S$ which means that $h^t : T^v_{\mathbf{sig}} \times T^s_{\mathbf{sig}} \to T^t_{\mathbf{sig}}$ and, for all $x \neq s$, h^x is the constant function $h^x : T^v_{\mathbf{sig}} \times T^x_{\mathbf{sig}} \to T^\varepsilon_{\mathbf{sig}} = \{(\,)\}$. It is customary to suppress all these meaningless constant functions in the primitive recursive definition; the type of g_F in Theorem 2.1 is then

$$g_F : T^v_{\mathbf{sig}} \times T^w_{\mathbf{sig}} \times T^{t^k}_{\mathbf{sig}} \to T^t_{\mathbf{sig}},$$

where k is the number of occurrences of s in w.

In [40] we have studied the analogue of primitive recursion in **sig**-d-algebras; we called this structural recursion in order to keep it apart from primitive recursion in term algebras.

The analogue of Theorem 2.1 then reads as follows.

Theorem 2.2 (Structural recursion). *Let* $\mathbf{sig} = \langle S, \Sigma \rangle$ *be a signature,* $\boldsymbol{A}$ *a* **sig**-d-*algebra,* $v \in S^*$ *and for all* $s \in S$ $u_s \in S^*$. *For* $w = w_1 \dots w_n \in S^*$ *let* $u_w := u_{w_1} \dots u_{w_n}$. *Then for every* Σ*-indexed family* g *of functions such that*

$$g_F : A^v \times A^w \times A^{u_w} \to A^{u_s} \quad \text{for } F \in \Sigma^{(w,s)},\ w = w_1 \dots w_n,$$

there is a unique S-indexed family h of operations $h = (h^s)_{s \in S}$, *where*

$$h^s : A^w \times A^s \to A^{u_s},$$

with the property that for all $x \in A^w$, $s \in S$, $y \in A^s$ *we have*

$$h^s(x, y) = g_F(x, z_1, \ldots, z_n, h^{w_1}(x, z_1), \ldots, h^{w_n}(x, z_n)),$$

where $d_A(y) = (F, z_1 \ldots z_n)$.

Proof. We will prove this theorem by using Recursion Theorem 1.17. Define a set X by

$$X := \bigcup_{s \in S} (A^s \times A^{u_s}).$$

(This is, of course, not the sort-preserving product. We denote its projections by pr_1 and pr_2. Elements of X are written as pairs (x_1, x_1), $x_1 \in A^s$, $x_2 \in A^{u_s}$.)

$(x_1, x_2) \in X$ is said to be of sort $s \in S$ whenever $x_1 \in A$ is of sort s. For all $w \in S^*$, X^w is isomorphic to $A^w \times A^{u_w}$. (Simple rearrangement of components.) If $f : A \to X$ is sort-preserving, then it can be written as a target product $f = [f_1; f_2]$ where $f_i = \mathrm{pr}_i \circ f$, $i \in [2]$. Then $f^w : A^w \to X^w$ can be described by

$$f^w(a_1, \ldots, a_r) := (f_1^w(a_1, \ldots, a_r), f_2^w(a_1, \ldots, a_r)).$$

After these preliminaries we first prove the existence of h. Let $x \in A^t$ arbitrary; x will remain fixed for this section.

For $F \in \Sigma^{(w,s)}$ define $F_{X,x} : X^w \to X^s$ by

$$F_{X,x}(y, z) := (F_A(y), g_F(x, y, z)) \quad (y \in A^w, z \in A^{u_w}).$$

Now X is a Σ-algebra, so Theorem 1.17 guarantees the existence of a mapping $\Phi_x : A \to X$ such that

$$\Phi_x(a) = F_{X,x}(\Phi_x^w(y)), \quad \text{where } d_A(a) = (F, y),\ F \in \Sigma^{(w,s)},\ y \in A^w.$$

Let $\Phi_{x,i} := \mathrm{pr}_i \circ \Phi_x$. According to the preliminary remarks this means that

(1) $\quad \Phi_x(a) = F_{X,x}(\Phi_{x,1}^w(y), \Phi_{x,2}^w(y))$.

Now we show that

(2) $\quad \Phi_{x,1} = \mathrm{id}_A$,

(3) $\quad \Phi_{x,2}(a) = g_F(x, y, \Phi_{x,2}^w(y))$ for $d_A(a) = (F, a)$, $F \in \Sigma^{(w,s)}$, $y \in A^w$.

This is proved by structural induction. Let $a \in A$ and assume that (2) holds for all $y <_{d_A} a$. (If $d_A(a) = (F, \varepsilon)$, then the induction hypothesis is void.)

Now let $d_A(a) = (F, y)$, $F \in \Sigma^{(w,s)}$, $y \in A^w$. Then

$$\begin{aligned}
\Phi_x(a) &= F_{X,x}(\Phi_x^w(y)) && \text{by Theorem 1.17,}\\
&= F_{X,x}(\Phi_{x,1}^w(y), \Phi_{x,2}^w(y)) && \text{by (1),}\\
&= F_{X,x}(y, \Phi_{x,2}^w(y)) && \text{by induction hypothesis,}\\
&= F_{X,x}(F_A(y), g_F(x, y, \Phi_{x,2}^w(w)) && \text{by definition of } F_{X,x},\\
&= (a, g_F(x, y, \Phi_{x,2}^w(y))) && \text{by the decomposition property of } d_A.
\end{aligned}$$

Now define, for all $s \in S$, $h^s : A^x \times A^s \to A^{u_s}$ by

$$h^s(x, y) := \Phi_{x,2}(y).$$

Obviously, $h = (h^s)_{s \in S}$ has the required properties.

Now we prove that h is uniquely determined. Let $h' = (h'^s)_{s \in S}$ be another family of operations with the properties of h. For all $x \in A^x$, h'^s induces an operation $h'^s_x : A^s \to A^{u_s}$ by $h'^s_x(y) := h'^s(x, y)$.

Now define $\Phi'_x : A \to X$ by

$$\Phi'_x(a) := (a, h'^s_x(a)).$$

This implies for $d_A(a) = (F, y)$, $F \in \Sigma^{(w,s)}$, $y = (y_1, \ldots, y_n) \in A^w$:

$$\begin{aligned}
\Phi'_x(a) &= (F_A(y), g_F(x, y, h'^{s_1}_x(y_1), \ldots, h'^{s_n}_x(y_n)))\\
&= F_{X,x}(y, h'^{s_1}_x(y_1), \ldots, h'^{s_n}_x(y_n))\\
&= F_{X,x}(\Phi'^w_x(y)).
\end{aligned}$$

Using Theorem 1.17 we then have $\Phi'_x = \Phi_x$ and consequently $h'^s = h^s$ for all $s \in S$. □

Note that, by the construction used in the proof of Theorem 1.17, there will be a minimally defined solution h as above, if some of the g_F are partial.

Definition 2.3. Under the premises of Theorem 2.2, h is said to be *generated by* g *using structural recursion*. Again, we want to express this fact by some functional similar to PR[ ] above. Since this functional will later (Section 3) appear as an operation in a special algebra, we must assign it a *type*. For this purpose we consider only one of the h^s ($s \in S$) at a time. Then, under the premises of Theorem 2.2, write

$$h^s = \mathrm{SRO}_A^{(x,s,u_s)}[g],$$

or alternatively, if $\Sigma = \{F_1, \ldots, F_n\}$, with a fixed order of operation symbols,

$$h^s = \mathrm{SRO}_A^{(x,s,u_s)}[g_{F_1}, \ldots, g_{F_n}].$$

The class of *structural recursive operations* on a **sig**-d-algebra $\boldsymbol{A}$ is the smallest class of functions containing all F_A for $F \in \Sigma$, all projections π_i^w, $w \in S^*$, $i \in [\lg(w)]$, and being closed under composition, target tupling and structural recursion.

At this point it seems necessary to study some examples explaining the mechanism of structural recursion as presented in Theorem 2.2.

Example 2.4. The simplest special case is primitive recursion on natural numbers. Let $S = \{s\}$ consist of a single sort; the sort $(s^k, s) \in D(S)$ is then abbreviated to (k). The natural numbers can be represented as a decomposition algebra

$$\boldsymbol{\omega} = (\omega; 0^{(0)}, suc^{(1)}; d),$$

with the obvious definitions of 0_ω and suc_ω and the trivial decomposition $d(n) =$ **if** $n = 0$ **then** $(0, \varepsilon)$ **else** $(suc, n-1)$.

In Theorem 2.2 let $v = s$, $u_s = s$ and $g_0 \in \mathrm{Ops}^{(1)}(w)$, $g_{suc} \in \mathrm{Ops}^{(3)}(w)$ be defined by

$$g_0 = \pi_1^{(1)}, \qquad g_{suc} = suc_\omega \circ \pi_3^{(3)}.$$

Then add := SRO(g) or add := SRO($\pi_1^{(1)}$, $suc_\omega \circ \pi_3^{(3)}$) defines the addition of natural numbers, since we have

$$\mathrm{add}(x, y) = g_0(x) = \pi_1^{(1)}(x) = x \quad \text{if } d(y) = (0, \varepsilon), \text{ i.e., } y = 0$$

and

$$\begin{aligned}\mathrm{add}(x, y) &= g_{suc}(x, y', \mathrm{add}(x, y')) = suc_\omega(\mathrm{add}(x, y')) \\ &= \mathrm{add}(x, y') + 1 \quad \text{if } d(y) = (suc, y'), \text{ i.e., } y = y' + 1.\end{aligned}$$

Example 2.5 (Primitive recursive word functions). There are several ways of defining an algebra corresponding to a free monoid over an alphabet Σ; we choose the following one:

Let $S = \{\text{letter}, \text{list}\}$, $\Sigma = \{empty^{(\to \text{list})}, add^{(\text{list letter} \to \text{list})}\}$. Define a **sig**-d-algebra $\boldsymbol{L}$ by

$$L^{\text{letter}} := \Sigma, \qquad L^{\text{list}} := \Sigma^*,$$

$$\mathrm{empty}_L(\,) := \varepsilon, \qquad add_L(w, a) := wa,$$

and the obvious decomposition d.

The concatenation of two words in Σ^* is described by

$$\mathrm{concat}(w, w') = \begin{cases} w & \text{if } w' = \varepsilon, \\ add_L(\mathrm{concat}(w, w''), a) & \text{if } w' = w''a. \end{cases}$$

In Theorem 2.2 let $v = \text{list}$, $u_{\text{letter}} = \varepsilon$, $u_{\text{list}} = \text{list}$. Consequently, we have to define

$$g_{empty} : L^{\text{list}} \to L^{\text{list}}$$

and

$$g_{add} : L^{\text{list}} \times L^{\text{list letter}} \times L^{\text{list}} \to L^{\text{list}}$$

with

$$g_{empty}(w) = w,$$

$$g_{add}(w, w'', a, \text{concat}(w, w'')) = \mathbf{add}_L(\text{concat}(w, w''), a),$$

so we let

$$\text{concat} := \text{SRO}(\pi_1^{\text{list}}, \mathbf{add}_L \circ [\pi_4^{\text{list}^2\text{letter list}}; \pi_3^{\text{list}^2\text{letter list}}]).$$

Example 2.6 (Tuple-valued structural recursion). Combining Examples 2.4 and 2.5, we consider lists (or words) over natural numbers. Let

$$S = \{\text{list}, \text{nat}\},$$

$$\Sigma = \{\mathbf{empty}^{(\varepsilon,\text{list})}, \mathbf{add}^{(\text{list nat,list})}, 0^{(\varepsilon,\text{nat})}, \mathbf{suc}^{(\text{nat,nat})}\}.$$

Let $\boldsymbol{A}$ be the disjoint union of $\boldsymbol{\omega}$ (Example 2.4) and $\boldsymbol{L}$ (Example 2.5) with sorts letter and nat identified and again with trivial decompositions. Let concat be defined as in Example 2.5. Let

$$v := \text{list}, \qquad u_{\text{list}} := \text{list nat}, \qquad u_{\text{nat}} := \text{list}.$$

Define

$$g_{empty} : A^{\text{list}} \to A^{\text{list nat}},$$

$$g_{add} : A^{\text{list}} \times A^{\text{list nat}} \times A^{\text{list letter list}} \to A^{\text{list nat}},$$

$$g_0 : A^{\text{list}} \to A^{\text{list}},$$

$$g_{suc} : A^{\text{list}} \times A^{\text{nat}} \times A^{\text{list}} \to A^{\text{list}},$$

by

$$g_{empty}(w) := (w, 0),$$

$$g_{add}(w_1, w_2, n_1, w_3, n_2, w_4) := (\mathbf{add}(w_4, n_1), \mathbf{suc}(n_2)),$$

$$g_0(w) := w,$$

$$g_{suc}(w_1, n, w_2) := \text{concat}(w_1, w_2).$$

Using Theorem 2.2, this gives uniquely determined operations

$$h^{\text{list}} : A^{\text{list}} \times A^{\text{list}} \to A^{\text{list nat}}$$

and

$$h^{\text{nat}} : A^{\text{list}} \times A^{\text{nat}} \to A^{\text{list}},$$

with the following properties:

(1) If $d(y) = (\mathbf{empty}, \varepsilon)$, i.e., $y = \varepsilon$, then

$$h^{\text{list}}(x, y) = g_{empty}(x) = (x, 0).$$

(2) If $d(y) = (\mathit{add}, y'n)$, i.e., $y = y'n$, then

$$h^{\mathrm{list}}(x, y) = g_{\mathit{add}}(x, y', n, h^{\mathrm{list}}(x, y'), h^{\mathrm{nat}}(x, n))$$
$$= (\mathit{add}(h^{\mathrm{nat}}(x, n), n), \mathit{suc}(\pi_2^{\mathrm{list\,nat}}(h^{\mathrm{list}}(x, y')))).$$

(3) If $d(y) = (0, \varepsilon)$, i.e., $y = 0$, then

$$h^{\mathrm{nat}}(x, y) = g_0(x) = x.$$

(4) If $d(y) = (\mathit{suc}, n)$, i.e., $y = n + 1$, then

$$h^{\mathrm{nat}}(x, y) = g_{\mathit{suc}}(x, n, h^{\mathrm{nat}}(x, n)) = \mathrm{concat}(x, h^{\mathrm{nat}}(x, n)).$$

It is obvious that

$$h^{\mathrm{nat}}(x, n) = x^{(n+1)} \quad \text{for all } n \in \omega,$$

and that

$$\pi_2^{\mathrm{list\,nat}}(h^{\mathrm{list}}(x, y)) = \lg(y),$$

where lg denotes the length function.

A careful consideration shows that for $y = n_1 n_2 \ldots n_m \in \omega^*$,

$$\pi_1^{\mathrm{list\,nat}}(h^{\mathrm{list}}(x, y)) = ((\ldots(x^{(n_m+1)} n_m)^{(n_{m-1}+1)} n_{m-1}) \ldots)^{(n_1+1)} n_1.$$

For instance, if

$$312 = \mathit{add}(\mathit{add}(\mathit{add}(\mathit{empty}, 3), 1), 2) \quad \text{and} \quad 5 = \mathit{add}(\mathit{empty}, 5)),$$

then

$$h^{\mathrm{list}}(5, 312) = (55553555531555535555315555535555312\ ,\ 3).$$

Using the schema of Theorem 2.2, we can define, for every $s \in S$, a single function

$$h^s : A^v \times A^s \to A^{u_s},$$

with a single structural recursion. The advantage of considering tuple-valued operations is, however, that it is as well possible to define $m \geq 1$ such operations in a single step.

Corollary 2.7 (Simultaneous recursion). *Let* $\mathbf{sig} = \langle S, \Sigma \rangle$ *be a signature,* $\boldsymbol{A}$ *a* **sig**-d-*algebra,* $m \in \omega$, $m \geq 1$, $v \in S^*$, *and for all* $s \in S$, $i \in [m]$ $u_{s,i} \in S^*$. *For* $w = w_1 \ldots w_n \in S^*$ *and* $i \in [m]$ *let* $u_{w,i} := u_{w_1,i} \ldots u_{w_n,i}$. *Then for every family* $g = (g_{F,i})_{F \in \Sigma, i \in [m]}$ *of functions such that* $F \in \Sigma^{(w,s)}$, $w = w_1 \ldots w_n$ *implies, for all* $i \in [m]$,

$$g_{F,i} : A^v \times A^w \times A^{u_{w_1,i}} \times \cdots \times A^{u_{w_n,i}} \to A^{u_{s,i}},$$

there is a unique family $h = (h_i^s)_{s \in S, i \in [m]}$ *of functions,*

$$h_i^s : A^v \times A^s \to A^{u_{s,i}}$$

with the property that, for all $x \in A^v$, $s \in S$, $y \in A^s$, *we have*

$$h_i^s(x, y) = g_{F,i}(x, z_1, \ldots, z_n, h_1^{w_1}(x, z_1), \ldots, h_1^{w_m}(x, z_n),$$
$$\vdots$$
$$h_m^{w_1}(x, z_1), \ldots, h_m^{w_m}(x, z_n))$$

where $d_A(y) = (F, z_1 \ldots z_n)$.

Proof. Define $u_s := u_{s,1} u_{s,2} \ldots u_{s,m}$ and

$$g_F := [g_{F,1}; \ldots; g_{F,m}] \quad \text{(target tupling)}.$$

Write the operations h^s, $s \in S$ generated by Theorem 2.2 as

$$h^s := [h_1^s; \ldots; h_m^s].$$

Then h_i^s has the required properties. □

3. Structural recursive schemata

In this section we want to develop a notation for structural recursive operations on **sig**-algebras which depends only on **sig**, but not as in the previous section on a special **sig**-algebra and its decomposition. For this purpose, we will define a *derived signature* $D(\mathbf{sig})$ where the previous operation symbols F and projections π occur as constants δF, $\delta\pi$ (δ standing for 'derived', of course) and where we have combinator symbols ¶ for target tupling, $ for substitution and SRS for structural recursion.

Definition 3.1. Let $\mathbf{sig} = \langle S, \Sigma \rangle$ be a signature such that for all $s \in S$ either

$$\Sigma^{(s)} = \{F_1^{(w_1,s)}, \ldots, F_{n_s}^{(w_{n_s},s)}\},$$

with a fixed order of enumeration, or

$$\Sigma^{(s)} = \emptyset.$$

Then define $D(\mathbf{sig}) := \langle S^* \times S^*, D(\Sigma) \rangle$, where $D(\Sigma)$ is the smallest set of $D(S^* \times S^*)$-sorted *derived operation symbols* with the properties

(1) $\delta F \in D(\Sigma)^{(\varepsilon,(w,s))}$ for all $F \in \Sigma^{(w,s)}$, $(w, s) \in D(S)$,

(2) $\delta\pi_i^w \in D(\Sigma)^{(\varepsilon,(w,w_i))}$ for all $w \in S^*$, $i \in [\lg(w)]$,

$\delta\pi_0^w \in D(\Sigma)^{(\varepsilon,(w,\varepsilon))}$ for all $w \in S^*$,

(3) for all $v = s_1 \ldots s_r \in S^*$ $(r \neq 0)$,

$$¶^{(w,v)} \in D(\Sigma)^{((w,s_1)(w,s_2)\ldots(w,s_r),(w,v))},$$

$$¶^{(w,\varepsilon)} \in D(\Sigma)^{(\varepsilon,(w,\varepsilon))},$$

(4) $\$^{(w,v,u)} \in D(\Sigma)^{((w,v)(v,u),(w,u))}$ for all $w, v, u \in S^*$,

(5) for all $w \in S^*$, $u: S \to S^*$ and $s \in S$,

$$\mathrm{SRS}^{(vs,u(s))} \in D(\Sigma)^{((vw_1u(w_1),u(s))\ldots(vw_{n_s}u(w_{n_s}),u(s)),(vs,u(s)))}$$

where $u(s_1 \ldots s_n) := u(s_1) \ldots u(s_n)$.

Then a *structural recursive* **sig**-*schema* (briefly: *srs-schema*) is an element of $T_{D(\mathbf{sig})}$.

For the definition of semantics of srs-schemata we introduce a derived $D(\mathbf{sig})$-algebra of a **sig**-d-algebra.

Definition 3.2. Let $\boldsymbol{A}$ be a **sig**-d-algebra.

The *derived* $D(\mathbf{sig})$-*algebra* $D(\boldsymbol{A})$ of $\boldsymbol{A}$ is defined by

$$D(\boldsymbol{A}) := \mathrm{Ops}(A)$$

and

$$\delta F_{D(\boldsymbol{A})}(\ ) := F_A, \qquad \delta\pi^w_{i_{D(\boldsymbol{A})}}(\ ) := \pi^w_i,$$

$$¶^{(w,v)}_{D(\boldsymbol{A})}(f_1, \ldots, f_r) := [f_1; \ldots; f_r],$$

$$\$^{(w,v,u)}_{D(\boldsymbol{A})}(f, g) := g \circ f,$$

$$\mathrm{SRS}^{(vs,u(s))}_{D(\boldsymbol{A})}(g_1, \ldots, g_{n_s}) := \mathrm{SRO}^{(vs,u(s))}[g_1, \ldots, g_{n_s}].$$

Then the *semantics* $[\![t]\!]^{\boldsymbol{A}}$ of a structural recursive **sig**-schema $t \in T_{D(\mathbf{sig})}$ is defined by

$$[\![t]\!]^{\boldsymbol{A}} := h_{D(\boldsymbol{A})}(t),$$

where $h_{D(\boldsymbol{A})}: T_{D(\mathbf{sig})} \to D(\boldsymbol{A})$ is the unique $D(\mathbf{sig})$-homomorphism.

For ease of notation, we write F instead of δF, π^w_i instead of $\delta\pi^w_i$, $[f_1; \ldots; f_r]$ instead of $¶^{(w,v)}(f_1, \ldots, f_r)$ and $g \circ f$ instead of $\$^{(w,v,u)}(f, g)$ when we write down srs-schemata. This procedure is justified by Definition 3.2.

Definition. The *structural recursive operations* of a **sig**-d-algebra $\boldsymbol{A}$ are precisely those functions $F: A^w \to A^t$ with $F = [\![t]\!]^{\boldsymbol{A}}$ for some $t \in T_{D(\mathbf{sig})}$.

In the following lemma, we will define some special srs-schemata in order to prove that certain operations are structural recursive.

Lemma 3.3 (Constant abstraction). *Let* **sig** *be a signature,* $x \in T^{(t,t)}_{D(\mathbf{sig})}$ *for some* $t \in S$. *Then for every* **sig**-d-*algebra* $\boldsymbol{A}$ *and* $w \in S^*$ *there is an srs-schema* $\mathit{abs}^{(w,t)}(x)$ *with the property*

$$[\![\mathit{abs}^{(w,t)}(x)]\!]^{\boldsymbol{A}} = [\![x]\!]^{\boldsymbol{A}}.$$

Proof. Define $\mathit{abs}^{(w,t)}(x) := x \circ \pi^w_0$. □

For signatures with 'inductive sorts' we can define constant abstractions without using denenerate projections (see [40]).

Lemma 3.4 (Definition by cases, 1). *Let* **sig** *be a signature,* $s, t \in S$, $\Sigma^{(s)} := \{F_1^{(w_1,s)}, \ldots, F_n^{(w_n,s)}\}$. *Then there is an srs-schema* $cond^{(t^n s,t)}$ *such that, for all* **sig**-d-*algebras* $\boldsymbol{A}$, $x_1, \ldots, x_n \in A^t$, $y \in A^s$,

$$[\![cond^{(t^n s,t)}]\!]^{\boldsymbol{A}}(x_1, \ldots, x_n, y) = x_i \quad \text{for } d_A(y) = (F_i, y_1 \ldots y_{k_i}).$$

Proof. Let $u : S \to S^*$ be defined by $u(s) := t$, $u(x) := \varepsilon$ for all $x \neq s$.
Define

$$cond^{(t^n s,t)} := \mathrm{srs}^{(t^n s,t)}(\pi_1^{t^n w_1 u(w_1)}, \ldots, \pi_n^{t^n w_n u(w_n)}). \qquad \square$$

Corollary 3.5 (Definition by cases, 2). *Let* **sig** *be a signature,* $s, t \in S$, $F \in \Sigma^{(s)}$. *Then there is an srs-schema* $if\text{-}F^{(stt,t)}$ *such that for all* $y \in A^s$, $x_1, x_2 \in A^t$,

$$[\![if\text{-}F^{(stt,t)}]\!]^{\boldsymbol{A}}(y, x_1, x_2) = \begin{cases} x_1 & \text{if } d_A(y) = (F, y_1 \ldots y_k), \\ x_2 & \text{otherwise.} \end{cases}$$

Proof. Let $\Sigma^{(s)} := \{F_1^{(w_1,s)}, \ldots, F_n^{(w_n,s)}\}$ and $F = F_i$ for some $i \in [n]$. Define

$$if\text{-}F^{(sst,t)} := cond^{(t^n s,t)} \circ [\alpha_1; \ldots; \alpha_n; \pi_1^{stt}],$$

where

$$\alpha_j = \begin{cases} \pi_2^{stt} & \text{if } i = j, \\ \pi_3^{stt} & \text{if } i \neq j. \end{cases} \qquad \square$$

The proof of this corollary contains implicitly a trivial corollary to Theorem 2.2: while in Theorem 2.2 the recursive argument is assumed to be the last argument, this needs not be true for arbitrary structural recursive operations, since we can use a target tupling of projections to obtain any desired order of arguments.

Example 3.6 (Structural recursive **sig**-schemata). Let $\mathbf{sig} = \langle S, \Sigma' \rangle$ with $\Sigma' := \Sigma \cup \{error^{(s,\mathrm{letter})}\}$ and S, Σ as in Example 2.5. (For sake of simplicity, we assume a special letter '*error*'; we will comment on error handling in a following section.)

Define some structural recursive **sig**-schemata as follows:

$$\mathrm{newq} := empty,$$

$$\mathrm{enq} := add,$$

$$\mathrm{front} := \mathrm{SRS}^{(\mathrm{list,letter})}(error, if\text{-}empty^{(\mathrm{list\,letter}^2)}),$$

$$\mathrm{deq} := \mathrm{SRS}^{(\mathrm{list,list})}(empty, if\text{-}empty^{(\mathrm{list\,letter\,list})} \circ$$
$$\circ [\pi_1^{\mathrm{list\,letter\,list}}; abs^{(\mathrm{list\,letter\,list})}(empty);$$
$$add \circ [\pi_3^{\mathrm{list\,letter\,list}}; \pi_2^{\mathrm{list\,letter\,list}}]].$$

These srs-schemata define the basic operations of a fifo queue: the trivial schemata newq and enq create an empty queue and add elements to a queue, first returns the first item in a queue, if the queue is not empty, and deq removes the first entry of a non-empty queue.

For every **sig**-d-algebra $\boldsymbol{A}$ and $q \in A^{\text{list}}$, we have

$$
\begin{aligned}
[\![\text{front}]\!]^{\boldsymbol{A}}(q) &= [\![\text{SRS}^{(\text{list,letter})}(\textit{error}, \textit{if-empty}^{(\text{list letter}^2)})]\!]^{\boldsymbol{A}}(q) \\
&= \text{SRO}^{(\text{list,letter})}(\textit{error}, [\![\textit{if-empty}^{(\text{list letter}^2)}]\!]^{\boldsymbol{A}})(q) \\
&= \begin{cases} \textit{error} & \text{if } d_A(q) = (\textit{empty}, \varepsilon), \\ [\![\textit{if-empty}^{(\text{list letter}^2)}]\!]^{\boldsymbol{A}}(q', a, [\![\text{front}]\!]^{\boldsymbol{A}}(q')) & \\ & \text{if } d_A(q) = (\textit{add}, q'a), \end{cases} \\
&= \begin{cases} \textit{error} & \text{if } d_A(q) = (\textit{empty}, \varepsilon), \\ a & \text{if } d_A(q) = (\textit{add}, q'a) \text{ and} \\ & \quad d_A(q') = (\textit{empty}, \varepsilon), \\ [\![\text{front}]\!]^{\boldsymbol{A}}(q') & \text{if } d_A(q) = (\textit{add}, q'a) \text{ and} \\ & \quad d_A(q') = (\textit{add}, q''b). \end{cases}
\end{aligned}
$$

and

$$
\begin{aligned}
[\![\text{deq}]\!]^{\boldsymbol{A}}(q) &= \text{SRO}^{(\text{list,list})}(\textit{empty}, [\![\textit{if-empty}^{(\text{list letter list})}] \\
&\quad \circ [\pi_1^{\text{list letter list}}; \textit{abs}^{(\text{list letter list})}(\textit{empty}); \\
&\qquad \textit{add} \circ [\pi_3^{\text{list letter list}}; \pi_2^{\text{list letter list}}]]\!]^{\boldsymbol{A}})(q) \\
&= \begin{cases} \textit{empty} & \text{if } d_A(q) = (\textit{empty}, \varepsilon), \\ ([\![\textit{if-empty}^{(\text{list letter list})}]\!]^{\boldsymbol{A}} \circ [\pi_1^{\text{list letter list}}; \textit{empty}; & \\ \quad [\![\text{add} \circ [\pi_3^{\text{list letter list}}; \pi_2^{\text{list letter list}}]]\!]^{\boldsymbol{A}})(q', a, [\![\text{deq}]\!]^{\boldsymbol{A}}(q'))] & \\ \qquad \text{if } d_A(q) = (\textit{add}, q'a), & \end{cases} \\
&= \begin{cases} \textit{empty} & \text{if } d_A(q) = (\textit{empty}, \varepsilon), \\ \textit{empty} & \text{if } d_A(q) = (\textit{add}, q'a) \text{ and} \\ & \quad d_A(q') = (\textit{empty}, \varepsilon), \\ \textit{add}([\![\text{deq}]\!]^{\boldsymbol{A}}(q'), a) & \text{if } d_A(q) = (\textit{add}, q'a) \text{ and} \\ & \quad d_A(q') = (\textit{add}, q''b). \end{cases}
\end{aligned}
$$

The example shows that the semantics of even simple srs-schemata is not perspicuous. In Section 4 we will therefore introduce an alternative, more readable notation.

4. Abstract software specifications

This section is a continuation of the thoughts we have already sketched in the Introduction.

Definition 4.1. An *abstract software specification* is given by

$$D = \langle \mathbf{spec}; \mathcal{O} \rangle,$$

where $\mathbf{spec} = \langle \mathbf{sig}, E \rangle$ is a finite equational specification and $\mathcal{O}$ is a finite set of structural recursive **sig**-schemata.

The operation symbols Σ in $\mathbf{sig} = \langle S, \Sigma \rangle$ are called *constructors*. A *model of D* is a **sig**-decomposition algebra $\boldsymbol{A}$ which satisfies E. A model $\boldsymbol{A}$ induces a set O of *admissible operations* given by the semantics of $\mathcal{O}$.

A model is called a *free model* if it is a canonical **sig**-term algebra isomorphic to $T_{\mathbf{spec}}$ with decomposition inherited from $T_{\mathbf{sig}}$.

The main characteristic of this approach which goes back to [39, 40] is that it splits a specification into two parts (a *structural component* and a *functional component*) distinguishing between *generating* and *defined operations* and not using equations for the specification of defined operations.

At this point it seems necessary to comment on 'primitive recursive equations'. In Theorem 2.2, the operation h generated by g using structural recursion is defined to be a unique operation which *satisfies a certain set of equations*. However, the semantics of the structural recursion differs significantly from the standard semantics of equations as described in Section 1. We illustrate this using the following example.

Example 4.2. Specification for finite sets of non-negative integers with cardinality. Let

$$S = \{\text{nat}, \text{set}\},$$

$$\Sigma = \{0^{(\varepsilon,\text{nat})}, \mathit{suc}^{(\text{nat},\text{nat})}, \mathit{create}^{(\varepsilon,\text{set})}, \mathit{insert}^{(\text{set nat},\text{set})}\},$$

$$E = \{\mathit{insert}(\mathit{insert}(s, n), n) = \mathit{insert}(s, n),$$
$$\mathit{insert}(\mathit{insert}(s, n), m) = \mathit{insert}(\mathit{insert}(s, m), n)\}$$

$$\mathcal{O} = \{\text{card}^{(\text{set},\text{nat})}\} = \{\text{SRS}^{(\text{set},\text{nat})}(0, \mathit{suc} \circ \pi_3^{\text{set nat}^2})\}.$$

According to Theorem 2.2, $\text{CARD} := [\![\text{card}^{(\text{set},\text{nat})}]\!]^{\boldsymbol{A}}$ is for every **sig**-d-algebra $\boldsymbol{A}$ the unique operation

$$\text{CARD}: A^{\text{set}} \to A^{\text{nat}},$$

which satisfies the equations

$$\text{CARD}(s) = \begin{cases} 0_A & \text{if } d_A(s) = (\mathit{create}, \varepsilon), \\ \mathit{suc}_A(\text{CARD}(s')) & \text{if } d_A(s) = (\mathit{insert}, s'a). \end{cases}$$

Note that $d_A(s) = (\mathit{insert}, s'a)$ implies both $\mathit{insert}_A(s', a) = s$ and $s' \neq s$ since d_A is well-founded. Therefore, if A^{nat} is isomorphic to non-negative integers, CARD computes the cardinality of a finite set. The well-founded decomposition mechanism is an essential premise for this property. The simple syntactic translation of the equations for CARD does not give the expected result:

Let **spec'** $= \langle S, \Sigma', E' \rangle$ where **spec** $= \langle S, \Sigma, E \rangle$ as above,

$$\Sigma' := \Sigma \cup \{\mathrm{card}^{(\mathrm{set,nat})}\},$$

$$E' := E \cup \{\mathrm{card}(\mathit{create}) = 0,$$
$$\mathrm{card}(\mathit{insert}(s, n)) = \mathit{suc}(\mathrm{card}(s))\}.$$

This is, of course, no correct specification for cardinality since the second equation for card does not take into account the case that n is already a member of s. Using initial algebra semantics, it is easily seen that the **spec**-reduct of $\mathrm{SEM}_{\mathbf{spec}'}$ is not isomorphic to $\mathrm{SEM}_{\mathbf{spec}}$, since in $\mathrm{SEM}_{\mathbf{spec}'}$ we have, for all $n \geq 1$, $[\mathit{suc}^n(0)] = [\mathit{suc}(0)]$. This means that the original data type has been destroyed; **spec'** is no enrichment [29] of **spec** contrary to the conjecture of [29] that every 'primitive recursive set of equations' specifies an enrichment. Using a final algebra semantics we still loose the uniqueness of card as defined by the above equations since there is a countable number of card-operations satisfying them.

Bergstra [2] has shown that there is no finite equational specification for card without hidden functions. For a discussion of the role of decompositions as opposed to hidden functions, see Remark 4.6.

Example 4.2 is also an example for the so-called *enrichment problem* encountered with equational specifications: if we add defining equations for a new operation to a specification that already contains some equations, then the new equations can interact with the old ones in such a way that the original data type is destroyed. The enrichment problem does not arise with structural recursive schemata (see Section 6).

We still have to give a semantics to Definition 4.1, i.e., we must explain which object is specified by $D = \langle \mathbf{spec}; \Phi \rangle$. A first approach could be to consider the class of all models of D as the semantics of D. We will, however, show that in principle it suffices to consider *free models*.

First, we prove an auxiliary lemma.

Lemma 4.3. *Let* $\mathbf{A}$ *be a model of* $D = \langle \mathbf{spec}; \Phi \rangle$. *Then there is a canonical term algebra* C *isomorphic to* $\mathrm{SEM}_{\mathbf{spec}}$ *and an injective mapping* $f: A \to C$ *such that*

(i) $h_A \circ f = \mathrm{id}_A$ *for the unique* $h_A: C \to A$;

(ii) $\mathbf{C}$ *(with decomposition inherited from* $\mathrm{SYS}_{\mathbf{spec}}$*) and* $\mathbf{A}$ *are compatible w.r.t.* f.

Proof. The proof follows the lines of Lemma 1.20 and the existence proof for CTA's in [29].

First define $f: A \to T_{\mathbf{sig}}$ by

$$f(a) := F f^n(\boldsymbol{b}) \quad \text{for } d_A(a) = (F, \boldsymbol{b}),$$

and $C := \{f(a) \mid a \in A\}$.

Then f is injective and satisfies (i) but is no homomorphism. C' has the following properties:

(a) it is decomposition closed, i.e., $Ft_1 \dots t_n \in C' \Rightarrow t_i \in C'$ for all $i \in [n]$;

(b) for some congruence classes $[t] \in \text{SEM}_{\text{spec}}$, C' contains a representative $t^* \in [t]$.

(Since $\boldsymbol{A}$ satisfies E, the images of different $a, b \in A$ are in different congruence classes of SEM_{spec}.)

We then have to give representatives for the other congruence classes of SEM_{spec} in a way that the properties of a CTA are preserved. This is done by an inductive procedure as in [29]; C' is protected from modifications by a different definition of term complexity.

Define $K : T_{\text{sig}} \to \omega$ by

$$K(F) := 0 \quad \text{for } F \in \Sigma^{(\varepsilon,s)},$$

$$K(Ft_1 \dots t_n) := \begin{cases} 0 & \text{if } Ft_1 \dots t_n \in C', \\ \max\{K(t_1), \dots, K(t_n)\} + 1 & \text{otherwise.} \end{cases}$$

By induction over $K(t)$ we define a family $\{C_n \mid n \in \omega\}$ of term sets with the properties

(1) $t \in C_n \Rightarrow K(t) \leq n$.

(2) If $t \in T_{\text{sig}}$ and if there is $\hat{t} \in [t]$ with $K(\hat{t}) \leq n$, then C_n contains a unique representative $t^* \in [t]$.

(3) If $Ft_1 \dots t_m \in C_n$, then $n = 0$ or $t_i \in C_{n-1}$ for all $i \in [m]$. If this has been accomplished, set $C := \bigcup \{C_n \mid n \in \omega\}$ and define

$$F_C(t_1, \dots, t_n) := (Ft_1 \dots t_n)^*.$$

It is easy to see that C is then a **sig**-CTA isomorphic to SEM_{spec}. By definition of f, it is trivial that $\boldsymbol{C}$ and $\boldsymbol{A}$ are compatible w.r.t. f.

We still have to define the C_n.

(1) Define $C_0 := C_0' \cup C'$ such that C_0 contains exactly one representative for every $F \in \Sigma^{(\varepsilon,s)}$, $s \in S$. This is done by taking into C_0' an arbitrary representative t_0 for every F which has no representative in C'.

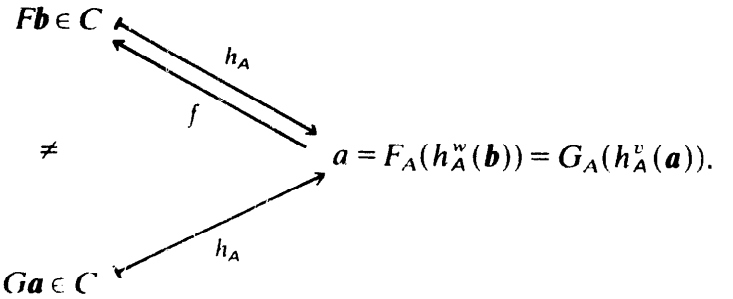
(2) Assuming that C_n satisfying (1)–(3) is already defined, define

$$T_{n+1} := \{[t] \in \text{SEM}_{\text{spec}} \mid (\exists t^* \in [t])\ K(t^*) = n+1, \text{ and } (\forall t' \in [t])\ K(t') \geq n+1\}.$$

In C_n we already have representatives for all $[t] \in T_n$. Let $[t] \in T_{n+1}$. Then by definition there is $t^* \in [t]$ with $K(t^*) = n+1$. Let $t^* = Ft_1 \dots t_m$ for some $F, t_1, \dots, t_m$. We then have $K(t_i) \leq n$ for all $i \in [m]$ implying that C_n contains already representatives $t_i^* \in [t_i]$. Then $Ft_1^* \dots t_n^*$ can be chosen as a representative for $[t] \in T_{n+1}$ satisfying properties (1)–(3). Define C_{n+1} as the union of C_n with representatives for all $[t] \in T_{n+1}$ obtained by this procedure. □

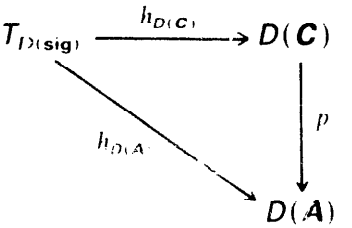
Remark 4.4. In Lemma 1.20 it was possible to have $f : A \to C$ as a homomorphism. We cannot expect this under the premises of Lemma 4.3, since $\boldsymbol{A}$ may be of smaller

cardinality than $\boldsymbol{C}$. It can therefore happen that C contains terms $\boldsymbol{Fb}$, $\boldsymbol{Ga}$ with $\boldsymbol{Fb} \neq \boldsymbol{Ga}$ but $F_A(h_A^w(b)) = a = G_A(h_A^v(a))$. However, we have *either* $d_A(a) = (F, \boldsymbol{b})$ or $d_A(a) = (G, \boldsymbol{a})$. Assuming, without loss of generality, $d_A(a) = (F, \boldsymbol{b})$, f has not the homomorphism property w.r.t. G:



Theorem 4.5. *Let $D = \langle \mathbf{spec}; \mathcal{C} \rangle$ be a specification, $\boldsymbol{A}$ a model of D. Then there is a free model $\boldsymbol{C}$ of D and a $D(\mathbf{sig})$-homomorphism $p: D(\boldsymbol{C}) \to D(\boldsymbol{A})$.*

Remark. Note that this implies commutativity of the following diagram:



where $h_{D(\boldsymbol{C})}$, $h_{D(\boldsymbol{A})}$ are the unique $D(\mathbf{sig})$-homomorphisms. Since these serve for the definition of the semantics $[\![s]\!]^{\boldsymbol{A}}$, $[\![s]\!]^{\boldsymbol{C}}$ of any srs-schema $s \in T_{D(\mathbf{sig})}$, Theorem 4.5 implies

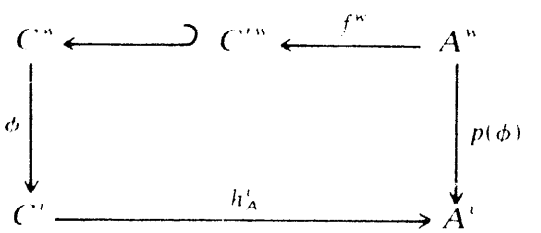
$$[\![s]\!]^{\boldsymbol{A}} = p \circ [\![s]\!]^{\boldsymbol{C}}.$$

Proof of Theorem 4.5. Let $\boldsymbol{C}$, C' and $f: A \to C$ be defined as in the proof of Lemma 4.3, $\mathrm{SRO}(C)$ the class of structural recursive operations on C.

Define $p: \mathrm{SRO}(C) \to \mathrm{Ops}(A)$ by

$$p(\phi) := h_A' \circ \phi \circ f^w \quad \text{for } \phi: C^w \to C'.$$

This means that for all $\phi \in \mathrm{SRO}(C)^{(w,r)}$ the following diagram commutes:



Furthermore, $h_A \circ f = \mathrm{id}_A$.

We have to show that ϕ is a $D(\mathbf{sig})$-homomorphism (see Definitions 3.1 and 3.2).

(1) For $F \in \Sigma^{(w,s)}$ we have

$$\begin{aligned} p(F_C) &= h_A \circ F_C \circ f^w \\ &= F_A \circ h_A^w \circ f^w \quad \text{because } h_A \text{ is a homomorphism} \\ &= F_A \circ \mathrm{id}_A \quad \text{by Lemma 4.3} \\ &= F_A, \end{aligned}$$

hence p respects the constants $\delta F \in D(\mathbf{sig})$.

(2) For $w \in S^*$, $i \in [\lg(w)]$, we have

$$\begin{aligned} p(\pi_i^w) &= h_A \circ \pi_i^w \circ f^w \\ &= h_A \circ f \circ \pi_i^w \quad \text{by definition of } f^w \\ &= \pi_i^w \quad \text{by Lemma 4.3.} \end{aligned}$$

$p(\pi_0^w) = \pi_0^w$ is trivial, hence p respects the constants $\delta\pi_i^w \in D(\mathbf{sig})$.

(3) For $\phi_i : C^w \to C^{s_i}$, $i = 1, \ldots, r$, $v = s_1 \ldots s_r$, we have

$$\begin{aligned} p([\phi_1; \ldots; \phi_r]) &= h_A^v \circ [\phi_1; \ldots; \phi_r] \circ f^w \\ &= [h_A \circ \phi_1 \circ f; \ldots; h_A \circ \phi_r \circ f] \\ &= [p(\phi_1); \ldots; p(\phi_r)], \end{aligned}$$

hence p respects the tupling operator $\P^{(w,v)}$.

(4) For $g_1 : C^v \to C^u$, $g_2 : C^w \to C^v$, we have

$$p(g_1) \circ p(g_2) = (h_A^u \circ g_1 \circ f^v) \circ (h_A^v \circ g_2 \circ f^w).$$

Here, we cannot simply eliminate $f^v \circ h_A^v$ in order to obtain $h_A^u \circ (g_1 \circ g_2) \circ f^w = p(g_1 \circ g_2)$ since in general $f \circ h_A \neq \mathrm{id}_C$.

For $c \in C^s$, we have $f(h_A(c)) = c$. We must therefore show that $(g_2 \circ f^w) : A^w \to C^{vs}$, whenever g_2 is structural recursive. This is done by induction over a term $t \in T_{D(\mathbf{sig})}$ with $[\![t]\!]^{\boldsymbol{C}} = g_2$. Note that the compatibility of $\boldsymbol{C}$ and $\boldsymbol{A}$ w.r.t. f is essential for this proof. Hence p respects the substitution operator $\$^{(w,v,u)}$.

(5) We denote by SRO_A, SRO_C the interpretation of SRS in $\boldsymbol{A}$ and $\boldsymbol{C}$, respectively. Again let $\Sigma^{(s)} = \{F_1^{(w_1,s)}, \ldots, F_{n_s}^{(w_{n_s},s)}\}$ as in Definition 3.1. We have to show that

$$p(\mathrm{SRO}_C^{(vs,u(s))}(\phi_1, \ldots, \phi_{n_s})) = \mathrm{SRO}_A^{(vs,u(s))}(p(\phi_1), \ldots, p(\phi_{n_s})).$$

First we have

$$p(\mathrm{SRO}_C^{(vs,u(s))}(\phi_1, \ldots, \phi_{n_s})) = h_A^{u(s)} \circ \mathrm{SRO}_C^{(vs,u(s))}(\phi_1, \ldots, \phi_{n_s}) \circ f^{vs}.$$

Here, $\mathrm{SRO}_C^{(vs,u(s))}(\phi_1, \ldots, \phi_{n_s})$ denotes the unique operation

$$H_C : C^v \times C^s \to C^{u(s)}$$

which, for $x \in C^r$, $y \in C^s$, satisfies

$$H_C(x, y) = \phi_i(x, y_1, \ldots, y_r, H_C^{u(s_1)}(x, y_1), \ldots, H_C^{u(s_r)}(x, y_r))$$

$$\text{for } d_C(y) = (F_i, y_1 \ldots y_r),\ y_i \in C^{s_i}.$$

For $a \in A^r$, $b \in A^s$ we then have

$$p(\mathrm{SRO}_C^{(rs,u(s))}(\phi_1, \ldots, \phi_{n_s}))(a, b) =$$

$$= h_A^{u(s)}(\mathrm{SRO}_C^{(rs,u(s))}(\phi_1, \ldots, \phi_{n_s}))(f^r(a), f(b))$$

$$= h_A^{u(s)}(H_C(f^r(a), f(b)))$$

$$= h_A^{u(s)}(\phi_i(f^r(a), c_1, \ldots, c_r, H_C^{u(s_1)}(f^r(a), c_1), \ldots, H_C^{u(s_r)}(f^r(a), c_r))$$

$$\text{for } d_C(f(b)) = (F_i, c_1 \ldots c_r),\ c_i \in C^{s_i}$$

$$= h_A^{u(s)}(\phi_i(f^r(a), f(b_1), \ldots, f(b_r),$$

$$H_C^{u(s_1)}(f^r(a), f(b_1)), \ldots, H_C^{u(s_r)}(f^r(a), f(b_r)))$$

$$\text{for } d_A(b) = (F_i, b_1 \ldots b_r), \text{ by definition of } f.$$

On the other hand, for $H_A = \mathrm{SRO}^{(rs,u(s))}(p(\phi_1), \ldots, p(\phi_{n_s}))$, we have

$$H_A(a, b) = \mathrm{SRO}_A^{(rs,u(s))}(p(\phi_1), \ldots, p(\phi_{n_s}))(a, b)$$

$$= p(\phi_i)(a, b_1, \ldots, b_r, H_A^{u(s_1)}(a, b_1), \ldots, H_A^{u(s_r)}(a, b_r))$$

$$\text{for } d_A(b) = (F_i, b_1 \ldots b_r),\ b_i \in A^{s_i}$$

$$= (h_A \circ \phi_i \circ f^{rs, u(s)})(a, b_1, \ldots, b_r, H_A^{u(s_1)}(a, b_1), \ldots, H_A^{u(s_r)}(a, b_r))$$

$$\text{by definition of } p$$

$$= h_A(\phi_i(f^r(a), f(b_1), \ldots, f(b_r)),$$

$$f^{u(s_1)}(H_A^{u(s_1)}(a, b_1)), \ldots, f^{u(s_r)}(H_A^{u(s_r)}(a, b_r))).$$

Therefore, it remains to show that

$$h_A^{u(s)}(H_C(f^r(a), f(b))) = H_A(a, b).$$

This is done by a structural induction.

(1) Let $d_A(b) = (F_i, \varepsilon)$, $F_i \in \Sigma^{(0,s)}$. Then the computations give the result

$$h_A^{u(s)}(H_C(f^r(a), f(b))) = h_A^{u(s)}(\phi_i(f^r(a))) = H_A(a, b).$$

(2) Let $d_A(b) = (F_i, b_1 \ldots b_r)$. Assume for all $i \in [r]$,

$$h_A^{u(s)}(H_C(f^r(a), f(b_i))) = H_A(a, b_i).$$

Then the rest of the proof follows by induction on a schema $t \in T_{D,\mathrm{sig}}$ with $[\![t]\!]^C = \phi_i$. Again, the compatibility of C and A w.r.t. f is essential to this proof. □

Remark 4.6. We want to comment again on Example 4.2, where we have presented a structural recursive specification of an operation which is not finitely specifiable using equations without hidden functions. The well-founded decomposition used for the semantics of structural recursive schemata could also be considered as some hidden function of a **sig**-d-algebra and, what is even worse, an implicit one in contrast with the explicitly specified hidden functions of an equational specification. Theorem 4.5, however, states that in principle it is enough to consider free models of specifications. Here, decomposition means term decomposition and is a trivial process. Moreover, if the equations contained in **spec** (for $D = \langle \mathbf{spec}; \mathcal{O} \rangle$) can be interpreted as rewrite rules (from left to right, e.g.) having the Church–Rosser and finite termination properties, then the set of normal forms of this rewriting system gives a really canonical method of obtaining a free model of D with trivial decomposition (see [59]; Schulz also shows that in principle it suffices to consider such sets of equations, as long as we want to specify computable data types and admit hidden functions.)

Of course, all free models of a specification $D = \langle \mathbf{sig}, E; \mathcal{O} \rangle$ are isomorphic as **sig**-algebras, even when they are different as subsets of $T_{\mathbf{sig}}$. Different choices of representatives for congruence classes relate to different methods for well-founded decomposition (cf. Theorem 1.12). Theorem 4.5 states that for any one model $\boldsymbol{A}$ of D with an arbitrary decomposition mechanism, we can always find a **sig**-CTA $\boldsymbol{C}$ in the isomorphism class of free models of D such that the unique term decomposition in $\boldsymbol{C}$ is compatible with decomposition in $\boldsymbol{A}$ and the semantics of a structural recursive schema in $\boldsymbol{A}$ can be derived from its semantics in $\boldsymbol{C}$. This justifies the following definition.

Definition 4.7. The abstract software module specified by $D = \langle \mathbf{sig}, E; \mathcal{O} \rangle$ is the **sig**-isomorphism class of free models of D.

Before we give further examples of abstract software specifications, we want to introduce a more readable notation for structural recursive schemata. We have already noted in Section 3 that the SRS combinator used for structural recursion is semantically very complex and not straightforward to understand.

Considering again the semantics of SRS (Definitions 2.3, 3.1 and 3.2) and the mechanism of structural recursion (Theorem 2.2), we develop the following notation: If $h = \mathrm{SRS}^{(vs, u(s))}(f_1, \ldots, f_n)$, then this means that h denotes a function $\bar{h}_{\boldsymbol{A}}$ of type $(vs, u(s))$ for every **sig**-d-algebra A, with the property

$$\bar{h}_{\boldsymbol{A}}(x, y) = \bar{f}_{i_A}(x, z_1, \ldots, z_n, H_{\boldsymbol{A}}^{v}(x, z_1), \ldots, H_{\boldsymbol{A}}^{w_n}(x, z_n))$$

$$\text{for } d_A(y) = \langle F, z_1 \ldots z_n \rangle, F_i \in \Sigma^{(w_1 \ldots w_n, s)}.$$

Here, $\bar{f}_{i_A}$ is the function denoted by f_i in $\boldsymbol{A}$. The implicit case distinction in this definition is now made explicit: Let $v = s_1 \ldots s_k$, $u(s) = t_1 \ldots t_m$. Then instead of

$$h = \mathrm{SRS}^{(vs, u(s))}(f_1, \ldots, f_n),$$

write

$$h(x_1 : s_1 ; \ldots ; y_k : s_k ; y : s) : t_1 \times \cdots \times t_m =$$

case y **of**

$F_1(y_1, \ldots, y_{m_1}) : T_1 ;$

$\vdots$

$F_n(y_1, \ldots, y_{m_n}) : T_n$

esac.

Here T_i is a term corresponding to the operation f_i; it may contain the parameter variables $x_1, \ldots, x_k$, the *formal predecessor variables* $y_1, \ldots, y_{m_i}$ introduced in every branch of the **case**-clause, all constructors F_i and recursive calls of h or other operations defined by similar clauses as long as the requirements of Theorem 2.2 (or Corollary 2.7) are met.

On this basis we have developed the Structural Recursive Definition Language SRDL which is described in [40, 41, 55]. SRDL has been implemented in order to allow preliminary testing of specifications before the design and implementation of corresponding programs. We will not further comment on SRDL in the present paper since here it is only used as a notation. Besides, it is rather self-explanatory if it is related to the formal development in the present paper. We only want to comment on a somewhat unusual notation for signatures in SRDL: The constructors $\{F_1^{(w_1, s)}, \ldots, F_n^{(w_n, s)}\}$ with target sort s are listed as

$$s = (F_1(w_1), \ldots, F_n(w_n)).$$

We have chosen this notation because it suggests that every element of sort s is either obtained as $F_1(\ldots)$ or $F_2(\ldots)$ or $\ldots F_n(\ldots)$ for arguments of appropriate sorts.

We will now give some examples of abstract software specifications in SRDL, simply reformulating previous examples from this paper.

Example 4.8. (a) Cf. Example 3.6:

module queue =
sorts letter, list;
constructors list = (empty, add(list letter));
letter = (error);
operations newq : list = empty;
enq(q : list ; i : letter) = add(q, i) ;
front(q : list) : letter = ,
case q **of**
empty : error;

```
                add(l, i): case l of
                              empty: i;
                              add(m, j): front(l)
                           esac
                esac;
        deq(q: list): list =
                case q of
                  empty: empty;
                  add(l, i): case l of
                                empty: empty;
                                add(m, j): add(deq(l), i)
                             esac
                esac
end
```

There are no reasonable models of 'queue' since a model is supposed to be generated by the empty set and our specification does not generate meaningful data of type letter. In fact, 'queue' is a typical example of a parameterized specification which will be treated in Section 8. Of course, we could at this point have specified a 'queue-of-nat' or something else, like in the following example.

(b) Cf. Example 4.2:

```
module set-of-nat =
sorts nat, set;
constructors nat = (0, suc(nat));
             set = (create, insert(set, nat));
variables s: set; n, m: nat;
equations insert(insert(s, n), n) = insert(s, n);
          insert(insert(s, n), m) = insert(insert(s, m), n);
operations card(s: set): nat =
                case s of
                  create: 0;
                  insert(t, n): suc(card(t))
                esac
end
```

There is, however, one point which deserves some attention. In the formal development where an srs-schema is represented by some term over $T_{D(\mathbf{sig})}$ there is no need for hidden functions at the operation level: since schemata can be nested to an arbitrary extent, a special auxiliary function needed for the specification of some operation can always occur *inside* the term representing this operation.

Example 4.9. We want to specify the multiplicative monoid of natural numbers, with operations unit (= '1') and multiplication. Using srs-schemata this can be

accomplished as follows (cf. Example 2.4):

$$D = (S, \Sigma, E; \mathcal{O}),$$

with

$$S = \{\text{nat}\}, \qquad \Sigma = \{0^{(\varepsilon,\text{nat})}, \mathit{suc}^{(\text{nat},\text{nat})}\},$$

$$E = \emptyset,$$

$$\mathcal{O} = \{\text{unit}^{(\varepsilon,\text{nat})}, \text{mult}^{(\text{nat}^2,\text{nat})}\},$$

where

$$\text{unit} = \mathit{suc} \circ O,$$

$$\text{mult} = \text{SRS}^{(\text{nat}^2,\text{nat})}(\mathit{abs}^{(\text{nat},\text{nat})}(O),$$

$$\text{SRS}^{(\text{nat}^2,\text{nat})}(\pi_1^{\text{nat}}, \mathit{suc} \circ \pi_3^{\text{nat}^3}) \circ [\pi_1^{\text{nat}^3}, \pi_3^{\text{nat}^3}]).$$

mult contains a nested occurrence of SRS; in fact, the inner SRS-term represents addition. Since addition calls itself recursively, it is not possible to translate this schema into SRDL without giving the addition a name and turning it into a separate definition. This means that we have something like

```
module monoid =
sorts nat;
constructors nat = (0, suc(nat));
operations unit : nat = suc(0);
           mult(x, y : nat) : nat = case y of
                                      0 : 0;
                                      suc(z) : let add(x, y : nat) : nat =
                                                 case y of
                                                   0 : x;
                                                   suc(z) : suc(add(x, z))
                                                 esac
                                               in add(x, mult(x, z))
                                    esac
end
```

Here, add appears as a hidden operation.

5. Correctness of specifications

For considering correctness concerns of any kind of object, we always need some reference with which this object should be compared. In the case of abstract software specifications, we can imagine three kinds of references:

(1) a special algebra,

(ii) a second abstract specification, and
(iii) a purely equational ('axiomatic') specification.

For (iii), there are mainly two motivations: first, it may be easier in early stages of design to state some known or expected properties of certain operations by equations instead of giving a recursive definition (see the comments in the Introduction); second, equational specifications will be involved in parameterized specifications (see Section 8).

It is one of the characteristics of our approach that specifications $D = \langle \mathbf{spec}; \mathcal{O} \rangle$ are split into a structural component **spec** describing the *data* (i.e., the carrier of some algebra) and a functional component $\mathcal{O}$ describing the *admissible operations* in terms of data elements. A model of D is, formally, a **spec**-algebra; but sometimes we want to view it rather as an $\mathcal{O}$-algebra. Hence the following definition.

Definition 5.1 (Operational algebra). Let $D = \langle \mathbf{spec}; \mathcal{O} \rangle$ be a specification, $\boldsymbol{A}$ a model of D. From $\boldsymbol{A}$ we derive an $\mathcal{O}$-algebra $A^{\mathcal{O}}$ with carrier A by associating with every schema $s \in \mathcal{O}$ its semantics in $\boldsymbol{A}$:

$$s_A := [\![s]\!]^{\boldsymbol{A}}.$$

$A^{\mathcal{O}}$ is called the *operational algebra* of $\boldsymbol{A}$.

For the formal treatment of SRDL-schemata as terms in $T_{D(\mathbf{sig})}$, there is nothing to add to this. If schemata are written in SRDL, then Definition 5.1 must be interpreted in such a way that the s on the left-hand side of the equation means the *name* of the schema and the s between the semantic brackets on the right-hand side means its *definition*. Of course, this requires that different schemata in SRDL have different names, which is a reasonable requirement anyway. We will not further distinguish between an SRDL schema and its name.

Definition 5.2 (Correctness w.r.t. a special algebra). Let $D = \langle \mathbf{spec}; \mathcal{O} \rangle$ be a specification, $\boldsymbol{B}$ a $\mathbf{spec}'$-algebra.

Then D is *correct w.r.t.* $\boldsymbol{B}$ iff $S' \subseteq S$, $\Sigma' \subseteq \mathcal{O}$ and there is a free model $\boldsymbol{C}$ of D such that

$$h_C(\mathrm{SYN}_{\mathbf{spec}'}) \cong \boldsymbol{B},$$

where $h_C : \mathrm{SYN}_{\mathbf{spec}'} \to C^{\mathcal{O}}$ is the unique $\mathbf{spec}'$-homomorphism to the operational algebra of $\boldsymbol{C}$.

This means that for every $b \in B$ there is exactly one corresponding abstract object in the $\mathbf{spec}'$-reduct of the data structure $\boldsymbol{C}$ and that for every operation F of $\boldsymbol{B}$ there is a corresponding schema in $\mathcal{O}$ which has the same behaviour. The specification may involve more sorts and more operations, as sometimes auxiliary constructions (hidden sorts and operations) are needed.

Of course the above definition can be slightly generalized by allowing injective functions $f: S' \to S, g: \Sigma' \to \mathcal{C}$ instead of insisting on subset relations. With appropriate renaming of operations or schemata this can be reduced to Definition 5.2.

We will not comment here on correctness w.r.t. a second abstract specification since this will be treated in the following section.

Definition 5.3 (Correctness w.r.t. an equational specification). Let $D = \langle \mathbf{spec}; \mathcal{O} \rangle$ be a specification, $\mathbf{spec}' = \langle S', \Sigma', E' \rangle$ an equational specification. D is called *correct w.r.t.* $\mathbf{spec}'$ iff $S' \subseteq S$, $\Sigma' \subseteq \mathcal{C}$ and there is a free model $\boldsymbol{C}$ of D such that

$$h_C(\mathrm{SYN}_{\mathbf{spec}'}) \cong \mathrm{SEM}_{\mathbf{spec}'},$$

where $h_C: \mathrm{SYN}_{\mathbf{spec}'} \to C^O$ is the unique $\mathbf{spec}'$-homomorphism.

There are many ways of proving the isomorphism required in Definition 5.3; one of them is the following.

Lemma 5.4. *Under the premises of Definition* 5.3, *let* $C' := h_C(\mathrm{SYN}_{\mathbf{spec}'}) \subseteq C^O$. *Then* C' *is isomorphic to* $\mathrm{SEM}_{\mathbf{spec}'} \Leftrightarrow$

(i) C' *satisfies the equations* E',

(ii) *for every* $c \in C'$ *there is some* $t \in \mathrm{SYN}_{\mathbf{spec}'}$ *with* $[\![t]\!]^C = c$, *and*

(iii) *there is a surjective mapping* $a: C' \to \mathrm{SEM}_{\mathbf{spec}'}$ (***abstraction mapping***).

Proof. '$\Rightarrow$' is trivial.

For '$\Leftarrow$', if C' satisfies E', then $C' \in Alg_{\mathbf{spec}'}$; therefore, the unique homomorphism $i: \mathrm{SEM}_{\mathbf{spec}} \to C'$ exists. If (ii) holds, then C' is generated by $\emptyset$ and i is therefore surjective. If additionally the surjective abstraction mapping exists, then i must be injective too, hence an isomorphism. □

For correctness proofs of specifications, it is therefore essential to prove the validity of certain equations between terms over structural recursive schemata. In Definition 1.8 we have defined the validity of equations in algebras; we use this for the following definition.

Definition 5.5. Let $D = \langle \mathbf{spec}; \mathcal{O} \rangle$, X a set of variables. The equation $t_1 = t_2$ for $t_1, t_2 \in T_\Sigma(X)$ is called *valid*: $\Leftrightarrow$ for all models $\boldsymbol{A}$ of D, A^O satisfies $t_1 = t_2$.

Next, we want to reformulate this definition in such a way that it suggests some method for proving the validity of equations in practice. Following Theorem 4.5, it is not necessary to consider all models of D; instead it suffices to consider all *free models*. Furthermore, the terms t_1, t_2 are best interpreted as *polynomial schemata* identifying special operations (polynomials, derived operations) in every algebra of appropriate signature. We quickly review the corresponding definitions:

Definition. (a) Let $\mathbf{sig} = \langle S, \Sigma \rangle$ be a signature, $X = \{x_i^s \mid s \in S,\ i \in \omega\}$ a standard alphabet of variables. Every $w = s_1 \ldots s_n \in S^*$ determines a finite non-empty alphabet $X_w = \{x_1^{s_1}, x_2^{s_2}, \ldots, x_n^{s_n}\} \subseteq X$. A term $t \in T_{\mathbf{sig}}(X_w)^s$ is called a **sig**-*polynomial schema of type* (w, s).

Define $\mathrm{POL}_{\mathbf{sig}}^{(w,s)} := T_{\mathbf{sig}}^{(w,s)}$.

(b) Let $t \in \mathrm{POL}_{\mathbf{sig}}^{(w,s)}$ be a polynomial schema, A a **sig**-algebra. For $a \in A^w$ define $\boldsymbol{a} : X_w \to A$ by $\boldsymbol{a}(x_i^s) := \pi_i^w(a)$. Then $\boldsymbol{a}$ can be uniquely extended to a homomorphism $\hat{\boldsymbol{a}} : T_{\mathbf{sig}}(X_w) \to A$. Define $\mathrm{POL}(t)_A : A^w \to A^s$ by $\mathrm{POL}(t)_A(a) := \hat{\boldsymbol{a}}(t)$. $\mathrm{POL}(t)_A$ is called the *polynomial over A defined by t.*

We can now reformulate Definition 5.5 as follows.

Lemma 5.6. *Let* $D = \langle \mathbf{spec}; \mathcal{C} \rangle$; X, X_w *as before. Then the equation* $t_1 = t_2$ *for* $t_1, t_2 \in T_\sigma(X_w)$ *is valid* $\Leftrightarrow$ *for all free models C of D we have*

$$\mathrm{POL}(t_1)_{C^\omega} = \mathrm{POL}(t_2)_{C^\omega}.$$

(Note that the type of polynomial schemata is not unique; every $t \in \mathrm{POL}_{\mathbf{sig}}^{(w,s)}$ is also in $\mathrm{POL}_{\mathbf{sig}}^{(w',s)}$ for all w' which somehow contain the letters of w. So Lemma 5.6 states no restriction on equations.)

It seems natural to prove the equation in Lemma 5.6 by structural induction. For the evaluation of either side of it, we can use the following equalities.

Corollary 5.7. (A) *For all* $s \in \mathcal{C}$, $st_1 \ldots t_n \in T_\sigma(X_w)$, $t \in C^w$ *we have*

$$\mathrm{POL}(st_1 \ldots t_n)_{C^\omega}(t) = [\![s]\!]^C(\mathrm{POL}(t_1)_{C^\omega}(t), \ldots, \mathrm{POL}(t_n)_{C^\omega}(t)).$$

(B) *If* $s \in \mathcal{C}$ *is defined by*

$$s(x_1 : s_1; \ldots; x_n : s_n) : s_{n+1} = \mathbf{case}\ x_i\ \mathbf{of}$$
$$F_1(y_1, \ldots, y_{m_1}) : t_1;$$
$$\vdots$$
$$F_k(y_1, \ldots, y_{m_k}) : t_k$$
$$\mathbf{esac}$$

(*in* SRDL *notation*), *then*

$$[\![s]\!]^C(c_1, \ldots, c_{i-1}, F_j b_1 \ldots b_r, c_{i+1}, \ldots, c_n) = t_j',$$

where t_j' *is obtained from* t_j *by replacing all* x_m *by* c_m ($m \in [n]$) *and all* y_k *by* b_k ($k \in [m_j]$).

Proof. (A) By the definition of polynomials, we have

$$\mathrm{POL}(st_1 \ldots t_n)_{C^\omega}(t) = \hat{\boldsymbol{t}}(st_1 \ldots t_n),$$

where $\hat{t}$ is the homomorphism $\hat{t}: T_\Theta(X_w) \to C^O$ induced by t. This implies

$$\hat{t}(st_1 \ldots t_n) = [\![s]\!]^C(\hat{t}(t_1), \ldots, \hat{t}(t_n)) = [\![s]\!]^C(\text{POL}(t_1)_{C^O}, \ldots, \text{POL}(t_n)_{C^O}).$$

(B) This is a direct consequence of Theorem 2.2 and the fact that C is a free model whose decomposition is unique term decomposition.

It seems appropriate to call (B) a *symbolic evaluation of s*. Lemma 5.6 together with Corollary 5.7 therefore allow the proof of validity of equations between schema terms by structural induction and symbolic evaluation.

Example 5.8. Consider the following specification (cf. Example 4.8(b)):

module set-of-nat =
sorts nat, set;
constructors nat = (0, suc(nat));
set = (create, ins(nat, set));
variables s: set; x, y: nat;
equations ins(x, ins(x, s)) = ins(x, s));
ins(x, ins(y, s)) = ins(y, ins(x, s));
operations union(s, t: set): set = **case** t **of**
create: s;
ins(x, t_1): ins(x, union(s, t_1))
esac
end

We want to show that the operation union is associative, i.e., that the equation

$$\text{union}(x_1^{\text{set}}, \text{union}(x_2^{\text{set}}, x_3^{\text{set}})) = \text{union}(\text{union}(x_1^{\text{set}}, x_2^{\text{set}}), x_3^{\text{set}})$$

is valid. (For sake of simplicity, we drop the superscript set.) We must therefore show that for any one free model C of set-of-nat and arbitrary $c_1, c_2, c_3 \in C^{\text{set}}$ we have

$$\text{POL}(\text{union}(x_1, \text{union}(x_2, x_3)))_{C^O}(c_1, c_2, c_3) =$$

$$= \text{POL}(\text{union}(\text{union}(x_1, x_2), x_3))_{C^O}(c_1, c_2, c_3)$$

We do this by case distinction and induction on c_3.

(1) $c_3 = \text{create}$.

Hypothesis: none.

Left-hand side:

$$\text{POL}(\text{union}(x_1, \text{union}(x_2, x_3)))_{C^O}(c_1, c_2, \text{create}) =$$

$$= [\![\text{union}]\!]^C(\text{POL}(x_1)_{C^O}(c_1, c_2, \text{create}),$$

$$\text{POL}(\text{union}(x_2, x_3))_{C^O}(c_1, c_2, \text{create}))$$

$$= [\![\text{union}]\!]^C(c_1, \text{POL}(\text{union}(x_2, x_3))_{C^O}(c_1, c_2, \text{create}))$$

by definition of polynomial

$$\overset{(A)}{=} [\![\text{union}]\!]^C(c_1, [\![\text{union}]\!]^C(\text{POL}(x_2)_C{}^o(c_1, c_2, \text{create}),$$

$$\text{POL}(x_3)_C{}^o(c_1, c_2, \text{create})))$$

$$= [\![\text{union}]\!]^C(c_1, [\![\text{union}]\!]^C(c_2, \text{create}))$$

$$\overset{(B)}{=} [\![\text{union}]\!]^C(c_1, c_2).$$

Right-hand side (for the computation of the right-hand side, we abbreviate by applying the definition of polynomials immediately wherever appropriate):

$$\text{POL}(\text{union}(\text{union}(x_1, x_2), x_3))_C{}^o(c_1, c_2, \text{create}) =$$

$$\overset{(A)}{=} [\![\text{union}]\!]^C(\text{POL}(\text{union}(x_1, x_2))_C{}^o(c_1, c_2, \text{create}), \text{create})$$

$$\overset{(B)}{=} \text{POL}(\text{union}(x_1, x_2))_C{}^o(c_1, c_2, \text{create})$$

$$\overset{(A)}{=} [\![\text{union}]\!]^C(c_1, c_2).$$

So (1) is proved.

(2) $c_3 = \text{ins}(x, m)$.

Hypothesis: for all $y < c_3$ assume

$$\text{POL}(\text{union}(x_1, \text{union}(x_2, x_3)))_C{}^o(c_1, c_2, y) =$$

$$= \text{POL}(\text{union}(\text{union}(x_1, x_2), x_3))_C{}^o(c_1, c_2, y).$$

(Here, $<$ is the decomposition ordering which in a free model coincides with the subterm relation.)

For the following computations we use a further abbreviation by allowing several applications of (A) in a single step.

Left-hand side:

$$\text{POL}(\text{union}(x_1, \text{union}(x_2, x_3)))_C{}^o(c_1, c_2, \text{ins}(x, m)) =$$

$$\overset{(A)}{=} [\![\text{union}]\!]^C(c_1, [\![\text{union}]\!]^C(c_2, \text{ins}(x, m))$$

$$\overset{(B)}{=} [\![\text{union}]\!]^C(c_1, \text{ins}(x, [\![\text{union}]\!]^C(c_2, m)))$$

$$\overset{(B)}{=} \text{ins}(x, [\![\text{union}]\!]^C(c_1, [\![\text{union}]\!]^C(c_2, m)))$$

$$\overset{(A)}{=} \text{ins}(x, \text{POL}(\text{union}(x_1, \text{union}(x_2, x_3)))_C{}^o(c_1, c_2, m)$$

$$= \text{ins}(x, \text{POL}(\text{union}(\text{union}(x_1, x_2), x_3))_C{}^o(c_1, c_2, m) \quad \text{by hypothesis.}$$

Right-hand side:

$$\text{POL}(\text{union}(\text{union}(x_1, x_2), x_3))_{C^\Omega}(c_1, c_2, \text{ins}(x, m)) =$$

$$\overset{(A)}{=} [\![\text{union}]\!]^{\boldsymbol{C}}([\![\text{union}]\!]^{\boldsymbol{C}}(c_1, c_2), \text{ins}(x, m))$$

$$\overset{(B)}{=} \text{ins}(x, [\![\text{union}]\!]^{\boldsymbol{C}}([\![\text{union}]\!]^{\boldsymbol{C}}(c_1, c_2), m))$$

$$\overset{(A)}{=} \text{ins}(x, \text{POL}(\text{union}(\text{union}(x_1, x_2), x_3)))_{C^\Omega}(c_1, c_2, m).$$

This finishes (2) and the whole proof.

The proof in Example 5.8 looks rather clumsy because of the two kinds of semantics (polynomial semantics $\text{POL}(\ )_{C^\Omega}$ and schema semantics $[\![\ ]\!]^{\boldsymbol{C}}$) and the ubiquitious conversions between them using (A). For practical purposes it seems best to drop all kinds of semantic brackets and to substitute the c_i directly for the x_i. This is shown in the following example where we denote several applications of (A) and a single application of (B) by '$\frown$'.

Example 5.9. We want to show that the operation union of Example 5.8 is commutative. For this we need a simultaneous induction on both arguments of union and also an application of one of the equations of set-of-nat; this is denoted by '$\equiv$'.

To show:

$$\text{union}(s, t) = \text{union}(t, s) \qquad \text{for arbitrary } s, t \in C^{\text{set}},\ \boldsymbol{C} \text{ a free model of set-of-nat.}$$

Induction hypothesis: For all u, v such that at least one of the conditions $u < s$ or $v < t$ is true, assume $\text{union}(u, v) = \text{union}(v, u)$.

(1) $t = \text{create}$.

Left-hand side:

$$\text{union}(s, \text{create}) \frown s.$$

Right-hand side:

(1a) $s = \text{create}$: trivial.

(1b) $s = \text{ins}(x, m)$:

$$\text{union}(\text{create}, \text{ins}(x, m)) \frown \text{ins}(x, \text{union}(\text{create}(\text{create}, m))$$

$$= \text{ins}(x, \text{union}(m, \text{create})) \quad \text{by hypothesis}$$

$$\frown \text{ins}(x, m) = s.$$

(2) $t = \text{ins}(x, m)$.

Left-hand side:

$$\text{union}(s, \text{ins}(x, m)) \curvearrowright \text{ins}(x, \text{union}(s, m)).$$

Right-hand side:

(2a) $s = \text{create}$:

$$\text{union}(\text{ins}(x, m), \text{create}) \curvearrowright \text{ins}(x, m) \curvearrowright \text{ins}(x, \text{union}(\text{cre}, m)).$$

(2b) $s = \text{ins}(y, n)$:

$$\text{union}(\text{ins}(x, m), \text{ins}(y, n)) \curvearrowright \text{ins}(y, \text{union}(\text{ins}(x, m), n))$$

$$= \text{ins}(y, \text{union}(n, \text{ins}(x, m))) \quad \text{by hypothesis}$$

$$\curvearrowright \text{ins}(y, \text{ins}(x, \text{union}(n, m))) \equiv \text{ins}(x, \text{ins}(y, \text{union}\ (n, m)))$$

$$= \text{ins}(x, \text{ins}(y, \text{union}(m, n))) \quad \text{by hypothesis}$$

$$\curvearrowright \text{ins}(x, \text{union}(m, s))$$

$$= \text{ins}(x, \text{union}(s, m)) \quad \text{by hypothesis.}$$

This completes the proof.

6. Extensions and enrichments

We begin this section by defining morphisms between abstract specifications. Morphisms are a means of comparing specifications; of course, morphisms between specifications should induce morphisms between the corresponding models. Since a specification consists of an equational specification and a set of recursion schemata, a morphism between abstract specifications will consist of a morphism of equational specifications and a morphism of schemata.

Morphisms of equational specifications were defined by Ehrich [14, 15]; we briefly recapitulate his definition.

Definition. Let $\mathbf{spec}_i = \langle S_i, \Sigma_i, E_i \rangle$, $i \in \{0, 1\}$, be equational specifications, $h: S_0 \to S_1$ and $h^*: S_0^* \to S_1^*$ the monoid homomorphism induced by h. Let $g: \Sigma_0 \to \Sigma_1$ be a mapping with the property

$$F \in \Sigma_0^{(w,s)} \Rightarrow g(F) \in \Sigma_1^{(h^*(w), h(s))}.$$

Then $f = (h, g)$ induces a mapping $\hat{f}: \text{SYN}_{\mathbf{spec}_0} \to \text{SYN}_{\mathbf{spec}_1}$ by

$$\hat{f}(Ft_1 \ldots t_n) := g(F)\hat{f}(t_1) \ldots \hat{f}(t_n), \quad n \geq 0.$$

f is called a *morphism* $f: \mathbf{spec}_0 \to \mathbf{spec}_1$ iff for all $t_1, t_2 \in \text{SYN}_{\mathbf{spec}_0}$ we have

$$t_1 \equiv_{E_0} t_2 \Rightarrow \hat{f}(t_1) \equiv_{E_1} \hat{f}(t_2).$$

f is called an *embedding* (of equational specifications) if h and g are both injective.

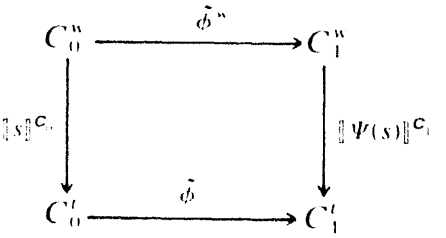
If $f: \mathbf{spec}_0 \to \mathbf{spec}_1$ is an embedding, then $f: \mathrm{SYN}_{\mathbf{spec}_0} \to \mathrm{SYN}_{\mathbf{spec}_1}$ can be made to a homomorphism by renaming the sorts and operations of $\mathbf{spec}_1$ according to the inverse of h and g. We will always assume such a renaming implicitly by speaking of $\hat{f}: \mathrm{SYN}_{\mathbf{spec}_0} \to \mathrm{SYN}_{\mathbf{spec}_1}$ as a homomorphism. Let $\mathrm{SYN}'_{\mathbf{spec}_1}$ be the $\langle h(S), g(\Sigma)\rangle$-reduct of $\mathrm{SYN}_{\mathbf{spec}_1}$; then $\hat{f}$ induces a homomorphism $\tilde{f}: \mathrm{SYN}_{\mathbf{spec}_0} \to \mathrm{SYN}'_{\mathbf{spec}_1}$. This is called the *homomorphism induced by f*. In fact, since $\hat{f}$ respects the equations, it is a homomorphism $\tilde{f}: \mathrm{SEM}_{\mathbf{spec}_0} \to \mathrm{SEM}'_{\mathbf{spec}_1}$.

Definition. Let $f = (h, g): \mathbf{spec}_0 \to \mathbf{spec}_1$ be an embedding of equational specifications. f is called an
- *extension* iff $\tilde{f}$ is an isomorphism,
- *equivalence* iff f is an extension and h, g are both bijective.

Thus extensions (on the specification level) correspond to embeddings in the algebraic sense (i.e., on the model level) and equivalences correspond to isomorphisms.

We are now ready to state the definition of a morphism between structural recursive specifications.

Definition 6.1 (Morphism of abstract software specifications). Let $D_i = \langle \mathbf{spec}_i; \mathcal{C}_i \rangle$, $i \in \{0, 1\}$, be abstract software specifications. A *morphism* $f: D_0 \to D_1$ is a pair $f = (\phi, \Psi)$ where $\phi: \mathbf{spec}_0 \to \mathbf{spec}_1$ is an embedding of equational specifications and $\Psi: \mathcal{C}_0 \to \mathcal{C}_1$ is a sort-preserving mapping such that for every pair $\boldsymbol{C}_0, \boldsymbol{C}_1$ of free models of D_0, D_1 that are compatible w.r.t. $\tilde{\phi}: \boldsymbol{C}_0 \to \boldsymbol{C}_1$, the following diagram commutes for all $s \in \mathcal{C}_0^{(w,t)}$:



Definition 6.2. Let $D_i = \langle \mathbf{spec}_i; \mathcal{C}_i \rangle$ be specifications, $i \in \{0, 1\}$; $f = (\phi, \Psi): D_0 \to D_1$ a morphism. f is called a
- *structural extension* $\Leftrightarrow$ ϕ is an extension,
- *structural equivalence* $\Leftrightarrow$ ϕ is an equivalence,
- *functional extension* $\Leftrightarrow$ Ψ is injective,
- *functional equivalence* $\Leftrightarrow$ Ψ is bijective.

The standard terminology of the literature (cf., e.g., [18, 29]) can be modelled in this context by the following definition.

Definition 6.3. Under the premises of Definition 6.2, f is called an
- *extension* $\Leftrightarrow$ f is both a structural and a functional extension,
- *enrichment* $\Leftrightarrow$ f is a structural equivalence and a functional extension.

Note, however, that there are differences between the properties of Definition 6.3 and the properties of enrichments/extensions in the equational framework; the main difference is that our enrichments/extensions are always *safe* in the sense of [29], i.e., they preserve the properties of the original specification. Before we show this, we want to come back to the previous section on specification correctness giving the supplementary definition:

Definition 6.4 (Correctness w.r.t. a structural recursive specification). Let $D_i = \langle \mathbf{spec}_i ; \mathcal{C}_i \rangle$ be specifications, $i \in \{0, 1\}$. D_1 is called *correct w.r.t.* D_0 iff there is a functional extension $f : D_0 \to D_1$.

Note that this definition includes the case that the underlying data structure for D_1 is completely altered w.r.t. D_0. Functional extension seems thus the concept which is best adapted for describing *implementations* in our approach. An implementation of D_0 by D_1 is just a functional extension $f : D_0 \to D_1$. We will devote a forthcoming paper to implementations and their properties.

Now we prove that our enrichments are safe enrichments.

Theorem 6.5 (Enrichments are safe). *Let* $D_i = \langle \mathbf{spec}_i ; \mathcal{C}_i \rangle$ *be specifications,* $i \in \{0, 1\}$, $f : D_0 \to D_1$ *an enrichment,* $\boldsymbol{A}$ *a model of* D_1. *Then*

(i) $\boldsymbol{A}$ *is also a model of* D_0, *and*

(ii) *for every* $s \in \mathcal{C}_0$, $[\![s]\!]^{\boldsymbol{A}} = [\![\Psi(s)]\!]^{\boldsymbol{A}}$.

Proof. By definition of structural equivalence, viz. equivalence of equational specifications, we can without loss of generality assume that

$$S_0 = S_1, \qquad \Sigma_0 = \Sigma_1, \qquad \mathrm{SEM}_{\mathbf{spec}_0} \simeq \mathrm{SEM}_{\mathbf{spec}_1}.$$

Hence it is trivial that $\boldsymbol{A}$ is also a model of D_0. Now let $\boldsymbol{C}$ be a free model of D_1 through which $\boldsymbol{A}$ factorizes according to Theorem 4.5. Then $\boldsymbol{C}$ is also a free model of D_0 and Definition 6.1 implies for all $s \in \mathcal{C}$ $[\![s]\!]^{\boldsymbol{C}} = [\![\Psi(s)]\!]^{\boldsymbol{C}}$. According to Theorem 4.5 this means also $[\![s]\!]^{\boldsymbol{A}} = [\![\Psi(s)]\!]^{\boldsymbol{A}}$. □

We have already commented on the conjecture on primitive recursive extensions, as stated in [29] (see Example 4.2). While this conjecture is not true for the equational treatment of primitive recursion, Theorem 6.5 shows that it *is* indeed true if primitive recursion is handled by structural recursive schemata.

It is a priori not possible to prove the analogue of Theorem 6.5 for extensions instead of enrichments; this is due to decompositions:

We can, without loss of generality, assume that

$$S_0 \subseteq S_1, \qquad \Sigma_0 \subseteq \Sigma_1,$$

$$\mathrm{SEM}_{\mathbf{spec}_0} \subseteq \mathrm{SEM}_{\mathbf{spec}_1} \quad (\text{a subalgebra}),$$

but nevertheless a model $\boldsymbol{A}$ of D_1 need not be a model of D_0. If $a \in \boldsymbol{A}$ is an element with $a = h_{\boldsymbol{A}}(t)$ for some $t \in \mathrm{SYN}_{\mathbf{spec}_0}$, then the decomposition $d_{\boldsymbol{A}}(a) = (F, a_1 \ldots a_n)$ does not necessarily give $F \in \Sigma_0$, since in $\boldsymbol{A}$ the operations of $\Sigma_1 \backslash \Sigma_0$ are also present. $\boldsymbol{A}$ is therefore not necessarily a $\mathbf{sig}_0$-decomposition algebra.

However, this can only occur if $\Sigma_1^{(h^*(w),h(s))}$ contains operation symbols which are not in $g(\Sigma_0^{(w,s)})$. This should be avoided anyway, if the Σ_i are meant as *constructors*, since the addition of new constructors to an existing sort amounts to a complete redefinition of that sort, meaning that we have to revise all srs-schemata with a recursion on that sort.

Definition. Let $D_i = \langle \mathbf{spec}_i ; \mathcal{C}_i \rangle$ be specifications, $i \in \{0, 1\}$, $f : D_0 \to D_1$ an extension. f is called a *clean extension* iff $\Sigma_1^{(h^*(w),h(s))}$ contains only operation symbols from $g(\Sigma_0^{(w,s)})$.

Corollary 6.6. *Clean extensions are safe.*

7. Exception handling

Error handling (or exception handling, as we prefer to call it) in the equational framework is a rather complicated task. The simple-minded exception handling fails because of problems similar to the problem encountered with enrichments of equational specifications: the 'error equations' interact with 'normal equations' in a way that the data type is destroyed: nothing but errors is left (see [29]).

There are, of course, correct methods for exception handling in equational specifications (see Section 10 for comments). None of these methods is simple; some of them, to the contrary, are rather awkward, and some require that once an error condition has been raised, it is propagated through all operation applications, thereby eliminating the possible specification of recovery from an exceptional situation.

It is our claim that in the structural recursive specification method the simple-minded exception handling is indeed possible and allows a rather flexible exception handling without assumptions on the propagation of exceptional states.

Definition 7.1. An *abstract software specification with exceptions* is

$$D = \langle \langle S, \Sigma, X, E \rangle ; \mathcal{C} \rangle,$$

where

- $\langle S, \Sigma, E \rangle$ is a finite equational specification,
- $X \subseteq \bigcup_{w \in S^*, s \in S} \Sigma^{(w,s)}$ is a set of *exceptions*, and
- $\mathcal{C}$ is a finite set of structural recursive $\langle S, \Sigma \rangle$-schemata.

There is almost nothing changed w.r.t. Definition 4.1; only we have marked a (possibly empty) set of constant constructors as denoting exceptional data elements. The pollution of the data structure by an infinity of meaningless data elements can be prevented by adding equations (see the following example).

Example 7.2. Suppose we want to specify natural numbers with an exceptional number 'undefined'. Using SRDL, this could be done by

module nat+ =
sorts nat;
constructors nat = (0, suc(nat));
exceptions nat = (undefined)
end

However, this does not mean that we have added a single data element 'undefined' to natural numbers; instead, every free model of nat+ will also contain different data elements corresponding to suc(undefined), suc(suc(undefined)) and so on.

If we add to nat+ the equation

suc(undefined) = undefined,

then it does what it is supposed to do. It is, however, also possible to add instead

suc(undefined) = 0,

which also inhibits pollution of nat but specifies an exception recovery.

Note that from the srs-schema point of view, exceptions are just treated as normal constructor constants; this implies that all structural recursive specifications must (in their case distinction) also take care of possible exceptions by explicitly specifying what has to be done in this case, be it error recovery or error propagation.

Example 7.3. In this example, nat+ is supposed to be the specification from Example 7.2 with the additional equation suc(undefined) = undefined. The **include**-statement of SRDL is a means of specifying a safe extension (see Section 6); it can be interpreted as being equivalent to the textual inclusion of all concepts of the included module in the present module at appropriate places.

module stack of nat=
include nat+(nat, 0, suc, undefined);
sorts stack;
constructors stack = (clear, push(stack, nat));
exceptions stack = (underflow, illegalstack);
variables *s*: stack; *n*: nat;
equations push(*s*, undefined) = illegalstack;
push(illegalstack, *n*) = illegalstack;
push(underflow, *n*) = push(clear, *n*);

```
operations pop(s:stack):stack=
                    case s of
                      clear, underflow:underflow;
                      push(t, n):t;
                      illegalstack:illegalstack
                    esac;
           top(s:stack):nat=
                    case s of
                      push(t, n):n;
                      otherwise undefined
                    esac
end
```

(The keyword **otherwise** collects, as in PASCAL, the remaining cases and is in the present context equivalent to 'clear, underflow, illegalstack'.)

Note that it is always necessary to check whether the equations relating to exception conditions are not inconsistent. With a view to the relatively small number of equations and the restricted form of right-hand sides, this is a much easier task than the usual one for equational specifications. In fact, the above set of equations is consistent, as can be easily seen. The third equation shows an exception recovery: a stack underflow is repaired by pushing something onto the stack. If we had instead specified

push(underflow, n) = clear,

then the system would have been inconsistent because of

clear = push(underflow, undefined) by the third equation,
= illegalstack by the first equation.

The second equation then guarantees that underflow and illegalstack are the only existing stacks. However, this 'crash' is limited to the sort stack and does not proliferate into sort nat, which would very probably happen if we were using an equational specification of top.

8. Parameterized specifications

It occurs rather frequently that, writing an abstract specification, we do not want to comment on special details of certain sorts or operations since at that point they do not matter. For instance, the definition of a set-of-something or a sequence-of-

something is independent of what this special thing happens to be. This situation is very similar to procedural abstraction in programming languages: they have formal parameters of which we only know the type (if at all?) and which can be actualized in a 'call' or an 'application' of the procedure.

Considering abstract software specifications and especially their use as a tool for design and development of software products, there is a second motivation for introducing parameterized specifications: we must be able to handle *incomplete specifications* where some concepts are left out because they are not yet well-understood at the present state of design but can only later be made precise. Such details can be considered as *parameters* of the specification.

It does, of course, not suffice to pass sorts as parameters to specifications, but we must also be able to pass operations (and exceptions) to a parameterized specification. Sometimes it is necessary that an operation supplied as a parameter has a special property (e.g., being symmetric on two arguments) or that there are special relations between several parameter operations (e.g., one being the inverse of the other.) Specifications of formal parameters are thus (at least) equational specifications.

Frequently, parameter operations and conditions refer to certain 'standard' specifications such as Boolean algebra, numbers, strings, etc. Instead of fixing a set of standard specifications, we allow the *parameter specification* and the *body specification* to have a *common part* which is referred to by an **include**-clause in the body specification.

It seems best to give some examples before stating the formal definition.

Example 8.1. Following the framework of Example 5.8, we want to specify finite sets over a parameter sort called data. Supposing that there is an *ordering relation* leq on data, the parameterized specification shall contain an operation max which returns a maximal element of a set w.r.t. the given leq; leq is therefore a parameter operation.

module set =
include boolean(bool, true, false, and, implies); }common part
parameters [**sorts** data;
operations default : data;
leq(data, data) : bool;
variables x, y, z : data;
equations leq(x, x) = true;
implies(and(leq(x, y), leq(y, z)),
leq(x, z)) = true] } parameter specification
sorts set;

```
constructors set = (empty, insert(set, data));
variables s: set; x, y: data;
equations insert(insert(s, x), x) = insert(s, x);
          insert(insert(s, x), y) = insert(insert(s, y), x);
operations max(s: set): data =
             case s of
               empty: default;
               insert(t, x): let lub(x, y: data): data =
                                 case leq(x, y) of
                                   true: y;
                                   false: x
                                 esac
                             in lub(x, max(t))
             esac
end
```

body specification

In addition to the predicate leq which comes from outside and is supposed to be at least reflexive and transitive, we also expect a special constant default which is returned as the maximum of the empty set. The signature of a boolean data type is imported by the **include**-clause. This is the *common part* of this specification; it can be referred to both in the parameter specification and in the body specification.

Example 8.2. A typical phenomenon of parameterized specifications are the so-called 'lifted operations'. In the following specification, an arbitrary unary operation f on a parameter sort data is lifted to list-of-data:

```
module list =
parameters [sorts data;                    } parameter
            operations f(data): data]      } specification
sorts list;
constructors list = (empty, atom(data), cons(list, data));
operations flist(l: list): list = case l of
                                    empty: empty;
                                    atom(a): f(a);
                                    cons(l1, a): cons(flist(l1), f(a))
                                  esac
end
```

body specification

The examples show that parameter specifications as well as body specifications are not always complete when viewed separately: parameter operations can take arguments and give results of parameter *and* common sorts; parameter equations can involve both common *and* parameter operations; the constructors of the body specification can involve common, parameter and body sorts. This is expressed in the following definition.

Definition 8.3. A *parameterized software specification* is given by

$D = (\mathbf{cp}, \mathbf{pspec}, \mathbf{spec}, \ldots)$,

where

$$\mathbf{cp} = \langle cS, c\Sigma\rangle, \qquad \mathbf{pspec} = \langle pS, p\Sigma, pE\rangle, \qquad \mathbf{spec} = \langle S, \Sigma, E\rangle,$$

such that the following conditions are satisfied:

(i) $cS \subseteq pS \subseteq S$, $c\Sigma \subseteq p\Sigma$;

(ii) **cp** is a finite signature, called the *common part*, **pspec** is a finite equational specification, called the *parameter specification*. The sorts in $pS - cS$ are called the *parameter sorts*; **spec** is a finite equational specification;

(iii) $\Sigma \cap p\Sigma = \emptyset$;

(iv) $s \in pS \Rightarrow \Sigma^{(s)} = \emptyset$;

(v) $\mathcal{C}$ is a finite set of structural recursive **sig'**-schemata not involving recursion on a sort $p \in pS$, for $\mathbf{sig'} = \langle S, \Sigma \cup p\Sigma\rangle$.

This definition states several restrictions of which some are due to our special philosophy of considering elements of Σ as formal constructors. We will briefly explain the conditions (i)–(v):

(i), (ii) See above.

(iii) An operation cannot at the same time be a formal parameter and a constructor of the body specification.

There are other approaches where the parameter specification **pspec** is supposed to be contained in **spec** (see, e.g., [19]).

This is appropriate for equational specifications but not for structural recursive ones, since parameter operations are not at the same conceptual level as constructors but rather belong to the (defined) operations, as we will see later.

(iv) If Σ would contain any constructor with result in parameter sort, then this would amount to a *complete redefinition* of that sort (cf. the remarks on clean extensions in Section 6). Note, however, that (iv) does *not* preclude the existence of *defined operations* with result in a parameter sort, like max in Example 8.1. In practice, this condition can be dropped by insisting that every **case**-definition ends with an **otherwise**-clause (see Example 7.3), which would catch also these new constructors, possibly yielding an exception. We have stated condition (iv) here because we did not want to develop a special theory for recursive definitions with **otherwise**.

(v) $\mathcal{C}$ cannot contain any structural recursive schema involving recursion on a parameter sort $s \in pS$. This would imply that we would have to know the formal constructors of sort s and their respective types, an information which is not supplied with the parameter specification.

Two further comments are necessary:

(1) Every specification $D = \langle \mathbf{spec}; \mathcal{C}\rangle$ according to Definition 4.1 can be considered as a parameterized specification $D = \langle\langle\emptyset, \emptyset\rangle; \langle\emptyset, \emptyset, \emptyset\rangle; \mathbf{spec}; \mathcal{C}\rangle$, so it suffices to consider only parameterized specifications.

(2) While in a non-parameterized specification every schema $s \in \mathcal{C}$ has a unique semantics, this is *not* true for parameterized specifications except for the case $p\Sigma = \emptyset$, since s can involve a call of a parameter operation $f \in p\Sigma$. In this case the semantics of s is not defined. (We will come back to this.)

Next we want to define when and how a parameterized specification D_0 can be *applied* to another specification D_1 and in what sense the actual parameter D_1 must satisfy the parameter conditions. A parameter assignment from D_0 to D_1 will assign a sort of D_1 to every parameter sort of D_0 and a defined operation of D_1 to every parameter operation of D_0 in such a way that the parameter equations of D_0 are satisfied. The semantics of the application of D_0 to D_1 using this parameter assignment will be some kind of union of D_0 and D_1, with the parameter sort or operation in D_0 replaced by its image under the parameter assignment. So the application of D_0 to D_1 will again be parameterized if D_1 is parameterized.

With a view to this intended semantics, we will require in the following definition (1) that every sort which D_0 and D_1 have in common has indeed the same structure in both specifications, and (2) that no parameter operation of D_1 coincides with a constructor of D_0.

Definition 8.4 (Parameter assignment). Let $D_i = \langle \mathbf{cp}_i ; \mathbf{pspec}_i ; \mathbf{spec}_i ; \mathcal{C}_i \rangle$, $i \in \{0, 1\}$ be parameterized specifications satisfying the following *compatibility conditions*:

(1) $s \in S_0 \cap S_1 \Rightarrow \Sigma_0^{(s)} = \Sigma_1^{(s)}$ and $E_0^s = E_1^s$,

(2) $\Sigma_0 \cap p\Sigma_1 = \emptyset$.

Then a pair $h = (f, g)$ of mappings

$$f : pS_0 \to S_1, \qquad g : p\Sigma_0 \to \mathcal{C}_1,$$

is a *parameter assignment* from D_0 to D_1 if

(3) g is compatible with f, i.e.,

$$F \in p\Sigma_0^{(w,s)} \Rightarrow g(F) \in \mathcal{C}_1^{(f^*(w), f(s))},$$

and

(4) for all equations $t_1 = t_2 \in pE_0$ and the $(p\Sigma_0 \cup c\Sigma_0)$-homomorphism $\hat{g} : T_{p\Sigma_0 \cup c\Sigma_0}(X) \to T_{c\Sigma_1}(X)$ induced by g, the equation

$$\hat{g}(t_1) = \hat{g}(t_2)$$

is valid.

Remark 8.5. The additional requirement $s \in pS_1 \Rightarrow \Sigma_0^{(s)} = \emptyset$ of [42] is no longer necessary because $s \in pS_1$, $s \notin S_0$ implies automatically $\Sigma_0^{(s)} = \emptyset$ whereas $s \in pS_1 \cap S_0$ implies $\Sigma_0^{(s)} = \Sigma_0^{(s)}$ by (1); but $\Sigma_1^{(s)} = \emptyset$ by Definition 8.3(iv).

Actually, Definition 8.4 goes one step farther than is permitted by Definition 5.5, where the equivalence of schema terms is only defined for non-parameterized specifications. It is therefore necessary to give a semantics to structural recursive schemata in parameterized specifications.

Lemma and Definition 8.6. Let $D = \langle \mathbf{cp}; \mathbf{pspec}; \mathbf{spec}; \mathcal{C} \rangle$ be a parameterized specification, $\boldsymbol{A}$ an $\langle S, \Sigma \rangle$-decomposition algebra which satisfies E, and $\boldsymbol{P}$ a **pspec**-algebra. Then the disjoint union $A \dot{\cup} P$ can be understood as a $\langle S, p\Sigma \cup \Sigma \rangle$-algebra $\boldsymbol{A} \dot{\cup} \boldsymbol{P}$ with a well-founded decomposition of all sorts in S. (Here, some of the restrictions in Definition 8.3 are essential.) In $\boldsymbol{A} \dot{\cup} \boldsymbol{P}$ every schema $s \in \mathcal{C}$ has a unique semantics which is called *semantics of s in* $\boldsymbol{A}$ *modulo* $\boldsymbol{P}$.

For the definition of validity of equations we again use initial algebras.

Definition 8.7. Let $D = \langle \mathbf{cp}; \mathbf{pspec}; \mathbf{spec}; \mathcal{C} \rangle$ be a specification, X, X_w as in Definition 5.5. Then $t_1 = t_2$ for $t_1, t_2 \in T_\Sigma(X_w)$ is called *valid* if for all canonical term algebras $C \cong T_{\mathbf{spec}}$ and the initial algebra T in $Alg_{\mathbf{pspec}}$ we have

$$\mathrm{POL}(t_1)_{C \dot{\cup} T} = \mathrm{POL}(t_2)_{C \dot{\cup} T}.$$

Note that Definition 5.5 is a special case of Definition 8.7 and that validity of equations can again be proved using structural induction and symbolic evaluation without having to consider all models of specifications.

Now we are ready to define the application of a parameterized specification using a parameter assignment.

Definition 8.8. Under the premises of Definition 8.4, f induces a mapping $\tilde{f}: S_0 \to S_0 \cup S_1$ by

$$\tilde{f}(s) := \begin{cases} f(s) & \text{for } s \in \mathrm{p}S_0, \\ s & \text{otherwise.} \end{cases}$$

Define $\bar{S}_0$ by $\bar{S}_0 := \tilde{f}(S_0)$. Define $\bar{\Sigma}_0$ by $F \in \Sigma_0^{(w,s)} \Rightarrow \bar{F} \in \bar{\Sigma}_0^{(\tilde{f}^*(w), \tilde{f}(s))}$. Then $\langle S_0 \cup S_1, \bar{\Sigma}_0 \rangle$ is a signature. In the same way, g induces a homomorphism $\tilde{g}$ from $(\Sigma_0 \cup \mathrm{p}\Sigma_0)$-schemata to $(\Sigma_0 \cup \mathcal{C}_1)$-schemata. Let $\bar{E}_0$ denote set of equations derived from E_0 by replacing every $F \in \Sigma_0$ by $\bar{F}$ as defined above. Then

$$D' = \langle \mathbf{cp}_1; \mathbf{pspec}_1; \bar{S}_0 \cup S_1, \bar{\Sigma}_0 \cup \Sigma_1, \bar{E}_0 \cup E_1; \tilde{g}(\mathcal{C}_0) \cup \mathcal{C}_1 \rangle$$

is a parameterized specification which is called the *application of* D_0 *to* D_1 *using* h (notation: $D_0^h D_1$).

Remark 8.9. There is a minor technical difficulty if SRDL is used: to guarantee that $\tilde{g}(\mathcal{C}_0) \cup \mathcal{C}_1$ is well-defined, we must assume that schemata in $\mathcal{C}_0$ and $\mathcal{C}_1$ do not have the same name unless they are syntactically equal. With the formal approach considering schemata as terms in $T_{D(\mathbf{sig})}$ this problem, of course, does not exist.

At this point, we want to comment on a slight generalization of parameter assignment and parameter semantics. If a specification has several parameters, we can think of situations where only some of them are actualized at a given time whereas others are left as parameters. Parameters assignments can thus be *partial*.

Definitions 8.4 and 8.8 can easily be modified to cover this more general situation. In any way, because $pS_1 \subseteq S_1$, a formal parameter sort p of D_0 can be handed over throughout the application of D_0 to D_1 by letting $f(p) := p'$ for some $p' \in pS_1$.

Example 8.10. Let set be the parameterized specification of Example 8.1, and let

```
module natle=
include boolean(bool, true, false);
sorts nat;
constructors nat = (0, suc(nat));
operations le(x, y:nat):bool = case x of
                                   0:true;
                                   suc(z):case y of
                                            0:false;
                                            suc(w):le(z, w)
                                          esac
                                 esac
end
```

The obvious parameter assignment from set to natle is given in the following SRDL definition:

```
module setofnat = apply set to natle by (data := nat; default := 0;
                                         leq := le)
end
```

Here, it remains to show that le really satisfies the equations in the parameter specification of set. This is easily done using structural induction and symbolic evaluation.

The definition of a parameter assignment actually forbids the application of a parameterized specification to itself. Considering Example 8.2, it would therefore not be possible to lift an operation f on data to an operation on lists of lists of data. But this problem can be solved by having two copies of the module 'list', where in one of them all sorts, constructors and operations have been renamed. This can also be justified by practical reasons: if we have lists of lists of data together with lists of data, then we must be able to distinguish whether a given list is just a list of data or a list of list of data. SRDL offers a renaming clause for use with parameterized specifications.

Example 8.11 (cf. Example 8.2)

```
module listlist =
parameters [sorts data;
            operations f(data):data]
```

apply list **to** list (dlist := list; dempty := empty; datom := atom;
dcons := cons; dflist := flist)
by (data := dlist; f := dflist)
end

This means that in the actual parameter specification 'list' the stated renamings have to be done before the parameter assignment can be defined.

According to Definition 8.8, the above specification of listlist is exactly equivalent to

module listlist =
parameters [**sorts** data;
operations f(data): data];
sorts dlist, list;
constructors dlist = (dempty, datom(data), dcons(dlist, data));
list = (empty, atom(dlist), cons(list, dlist));
operations dflist(l: dlist): dlist = **case** l **of**
dempty: dempty;
datom(a): f(a);
dcons($l1$, a): dcons(dflist($l1$), f(a))
esac;
flist(l: list): list = **case** l **of**
empty: empty;
atom(a): dflist(a);
cons($l1$, a): cons(flist($l1$), dflist(a))
esac
end

An important property of parameterized specifications is the *preservation of actual parameters.*

Corollary 8.12. *If* $D_1 = \langle\langle\emptyset, \emptyset\rangle; \langle\emptyset, \emptyset, \emptyset\rangle; \mathbf{spec}_1; \mathcal{O}_1\rangle$ *and if* h *is a parameter assignment from* D_0 *to* D_1, *then* $D_0{}^hD_1$ *is a clean extension of* D_1.

Proof. This follows directly from the compatibility conditions in Definition 8.4 and the restrictions in Definition 8.3. Note that the parameter conditions cannot lead to a collapse of the common part; this is due to the definition of validity for parameter conditions. □

We could not formulate this result for the general case of $D_1 = \langle\mathbf{cp}_1; \mathbf{pspec}_1; \mathbf{spec}_1; \mathcal{O}_1\rangle$ because we have not defined a notion of extension for parameterized specifications.

An obvious question in this context is whether the application of parameterized specifications is associative. Precisely formulated, this question is the following:

Given $D_i = \langle \mathbf{cp}_i ; \mathbf{pspec}_i ; \mathbf{spec}_i ; \mathcal{O}_i \rangle$, $i \in \{0, 1, 2\}$, and parameter assignments

$$h_1 : D_0 \to D_1 \quad \text{and} \quad h_2 : D_0{}^{h_1}D_1 \to D_2,$$

do there exist parameter assignments

$$h_2' : D_1 \to D_2 \quad \text{and} \quad h_1' : D_0 \to D_1^{h_2'}D_2$$

such that

$$(D_0^{h_1}D_1)^{h_2}D_2 \equiv D_0^{h_1'}(D_1^{h_2'}D_2),$$

for a reasonable notion $\equiv$ of specification equivalence?

We will develop sufficient conditions for this question to be answered positively with $\equiv$ denoting equality of specifications.

The compatibility conditions for D_0 and $D_1^{h_2}D_2$ read:

(1.1) $s \in (S_0 \cap S_1) \cup (S_0 \cap S_2) \Rightarrow \Sigma_0^{(s)} = \bar{\Sigma}_1^{(s)} \cup \Sigma_2^{(s)}$ and $E_0^s = (\bar{E}_1^s \cup E_2^s)$.

(1.2) $\Sigma_0 \cap \mathrm{p}\Sigma_2 = \emptyset$.

We know that $D_0^{h_1}D_1$ and D_2 are compatible, which means that

(2.1) $s \in (S_0 \cap S_2) \cup (S_1 \cap S_2) \Rightarrow (\bar{\Sigma}_0^{(s)} \cup \Sigma_1^{(s)}) = \Sigma_2^{(s)}$ and $(\bar{E}_0^s \cup E_1^s) = E_2^s$.

(2.2) $(\bar{\Sigma}_0 \cup \Sigma_1) \cap \mathrm{p}\Sigma_2 = \emptyset$.

Furthermore, we know that D_0 and D_1 are compatible, hence

(3.1) $s \in S_0 \cap S_1 \Rightarrow \Sigma_0^{(s)} = \Sigma_1^{(s)}$ and $E_0^s = E_1^s$.

(3.2) $\Sigma_0 \cap \mathrm{p}\Sigma_1 = \emptyset$.

Now we assume that *shared sorts are parameter-free*, which means: if S_0 and S_1 or S_0 and S_2 or S_1 and S_2 have a common sort s, then no constructor of sort s involves arguments of parameter sorts. In this case, we have $\bar{\Sigma}_i^{(s)} = \Sigma_i^{(s)}$, $\bar{E}_i^s = E_i^s$ for $i \in \{0, 1, 2\}$.

We want to prove that D_1 and D_2 are compatible, i.e.,

(4.1) $s \in S_1 \cap S_2 \Rightarrow \Sigma_1^{(s)} = \Sigma_2^{(s)}$ and $E_1^s = E_2^s$.

(4.2) $\Sigma_1 \cap \mathrm{p}\Sigma_2 = \emptyset$.

(4.2) directly follows from (2.2).

Consider now (4.1). Let $s \in S_1 \cap S_2$. There are two cases:

(a) $s \notin S_0$. Then $\Sigma_0^{(s)} = \emptyset$, $E_0^s = \emptyset$ and (2.1) implies (4.1).

(b) $s \in S_0$. Then $s \in S_0 \cap S_1 \cap S_2$ and combination of (2.1) and (3.1) gives (4.1).

Now we come back to (1.1) and (1.2). Again, (2.2) implies (1.2). Consider (1.1). Let $s \in (S_0 \cap S_1) \cup (S_0 \cap S_2)$. There are three cases:

(a) $s \in S_0 \cap S_1 \cap S_2$. The combination of (3.1) and (4.1) gives (1.1).

(b) $s \in S_0 \cap S_1$, $s \notin S_2$. Then $\Sigma_2^{(s)} = \emptyset$, $E_2^s = \emptyset$ and (3.1) gives (1.1).

(c) $s \in S_0 \cap S_2$, $s \notin S_1$. Then $\Sigma_1^{(s)} = \emptyset$, $E_1^s = \emptyset$ and (1.1) is equivalent to the following condition:

$$s \in S_0 \cap S_2 \Rightarrow \Sigma_0^{(s)} = \Sigma_2^{(s)} \text{ and } E_0^s = E_2^s$$

which has to be stated as an additional requirement on D_0 and D_2.

This leads to the following result.

Lemma 8.13. *Let* $D_i = \langle \mathbf{cp}_i ; \mathbf{pspec}_i ; \mathbf{spec}_i ; \mathcal{O}_i \rangle$, $i \in \{0, 1, 2\}$ *and let*

$$h_1 : D_0 \to D_1 \quad \text{and} \quad h_2 : D_0^{h_1} D_1 \to D_2$$

be parameter assignments. Let shared sorts be parameter-free and D_0 *and* D_2 *satisfy the condition*

$$s \in S_0 \cap S_2 \Rightarrow \Sigma_0^{(s)} = \Sigma_s^{(s)} \text{ and } E_0^s = E_2^s.$$

Then h_2 *is a parameter assignment from* D_1 *to* D_2 *and* h_1 *from* D_0 *to* $D_1^{h_2} D_2$ *and we have*

$$(D_0^{h_1} D_1)^{h_2} D_2 = D_0^{h_1} (D_1^{h_2} D_2).$$

9. Examples

In this section, we will present some further examples of abstract software specifications, at the same time giving some hints on their practical use.

As a supplement to Example 4.8(b), we give a parameterized specification of a FIFO queue.

Example 9.1

```
module fifo =
parameters [sorts item;
            exceptions error:item];
sorts list;
constructors list = (empty, add(list, item));
exceptions list = (illegalqueue, underflow);
variables g:list; i:item;
equations add(q, error) = illegalqueue;
          add(illegalqueue, i) = illegalqueue;
          add(underflow, i) = add(empty, i);
operations newq:list = empty;
           enq(q:list; i:item) = add(q, i);
           front(q:list):item = case q of
                                  add(l, i): case l of
                                               empty: i;
                                               add(m, j): front(l);
                                               otherwise error
                                             esac;
                                  otherwise error
                                esac;
```

```
deq(q:list):list = case q of
                       empty, underflow:underflow;
                       illegalqueue:illegalqueue;
                       add(l, i):case l of
                                     empty:empty;
                                     add(m, j):add(deq(l), i);
                                     otherwise illegalqueue
                                 esac
                   esac
end
```

This is a parameterized specification which is complete w.r.t. exceptions. The suggested methodology in writing this kind of specifications is clear:

(1) Imagine the underlying data structure; design a set of operations generating all data elements (constructors).

(2) Specify exceptional situations.

(3) Think of special properties of constructors and relationships between them. If there ought to be some, specify them by equations. Specify the reaction of constructors to exceptional data. If necessary, go back to (2) adding new exceptions.

(4) Specify the admissible operations using structural recursive schemata. If the need for new exceptions arises, go back to (2). Note that the defined operations are the only things which a module exports (i.e., they can be used outside the module). So if some constructors are to be used from outside the specification, they have to be added as trivial schemata.

Constructors are therefore in general hidden operations which are not accessible from the outside. This allows restriction on constructor application in an easy and straightforward way. As an example, we present the specification of a bounded priority queue; natlt is here a specification similar to Example 8.10 with an additional exception 'undefined' and a Boolean-valued operation lt ('less than').

Every entry is provided with a priority which (for example purposes) is supposed to be a natural number. The operation front has to return the 'first' entry in the queue with a maximal priority. (There may be several items with an equally high priority in the queue.) Note that this requires the examination of the whole queue in order to find this item. (We will comment on more efficient implementations after the example.) In the same way, deq has to delete this special item from a queue. We do this by processing the queue twice: in a first pass, the index of the first entry with maximal priority is computed, in a second step, the item with this index is removed. Again, this seems a very costly procedure.

Example 9.2

```
module priorityqueue =
include natlt(nat, 0, suc, undefined, bool, true, false, lt):
```

```
parameters [sorts item;
            operations limit:nat;
            exceptions error:item];
sorts list;
constructors list = (empty, add(list, item, nat));
exceptions list = (illegalqueue, underflow, overflow);
variables q:list; i:item; n:nat;
equations add(q, error, n) = illegalqueue;
          add(q, i, undefined) = illegalqueue;
          add(underflow,i, n) = add(empty, i, n);
          add(overflow, i, n) = overflow;
operations newq:list = empty;
           length(q:list):nat = case q of
                                  empty:0;
                                  add(l, i, prio):suc(length(l));
                                  otherwise undefined
                                esac;
           enq(q:list; i:item; prio:nat):list
                 = case lt(length(q), limit) of
                     true:add(q, i, prio);
                     false:overflow;
                     otherwise illegalqueue
                   esac;
           front(q:list):item = let highest(l:list):nat×item =
                                  case l of
                                    empty:(0, error);
                                    add(l1, il, pl):
                                      let lub(x:nat; ix:item; y:nat; iy:item)
                                            nat×item =
                                        case lt(x, y) of
                                          true:(y, iy);
                                          false:(x, ix);
                                          otherwise (undefined, error)
                                        esac
                                      in lub(pl, il, highest(l1));
                                    otherwise (undefined, error)
                                in p2 (highest(q));
                (* p2 denotes the projection on the second component *)
           deq(q:list):list = let index(l:list; pos:nat):nat×nat =
                                case l of
                                  empty:(0, pos);
                                  add(l1, i1, pr1):
```

```
let lub(x, ix, y, iy: nat): nat × nat =
  case lt(x, y) of
    true: (y, iy);
    false: (x, ix)
  esac
in lub(p1, suc(pos),
       index(l1, suc(pos)));
  otherwise (undefined, undefined)
esac
in let remove(q: list; n: nat): list =
        case q of
          empty: underflow;
          add(l, i, pr): case n of
                           0: q;
                           suc(n1): add(remove(l, n1),
                                        i, pr);
                           otherwise illegalqueue
                         esac;
          otherwise illegalqueue
        esac
      in
      remove(q, p2(index(q, 0)))
end
```

Comments. This specification shows at the same time the advantages and disadvantages of the constructive specification technique based on structural recursion. Let us comment on the *disadvantages* first. Obviously, the definitions of front and deq are surprisingly complex when compared with their definition in terms of natural language. It is obvious that they can be much easier specified in a non-constructive way, e.g., using predicate calculus. The reason for this is, of course, that we must not only specify *what* front and deq do, but rather *how* these operations could in principle be computed. The effort which goes into the constructive specification, however, is probably lost because of the striking inefficiency of this specification. An implementation should never exactly follow the 'algorithms' of this specification whose main flaws are the following:

(1) The length of a queue is explicitly computed every time before a new item is entered into the queue.

(2) The two passes required by deq for removing an item from the queue seem excessive.

An implementation would probably have an integer variable containing the length of a queue and being updated by enq and deq and would use a different algorithm for enq which ensures that the item with highest priority is always the first item which is found when we search the queue. This gives efficient and short specifications for front and deq.

On the other hand, it is an *advantage* of the structural recursive specification technique that it forces us to unambiguously describe the effect of operations by a case analysis on the underlying data structure. This gives hope to the conjecture that no special case has been left out unintentionally. Efficiency concerns should not be a guideline in writing specifications. In most cases, efficiency can be added later by using an alternative data representation. We should all be familiar with this kind of phenomena, see for instance the primitive recursive definition of addition of natural numbers by

$$\begin{aligned}\mathrm{add}(x, y:\mathrm{nat}):\mathrm{nat} = \textbf{case } y \textbf{ of}\\ 0 : x\,;\\ \mathrm{suc}(z) : \mathrm{suc}(\mathrm{add}(x, z))\\ \textbf{esac}\end{aligned}$$

in Example 4.8. The evaluation of add(1, 3 000 000) according to this specification requires 3 million recursive calls of add whereas in everday arithmetic we 'compute' the result in a single step, because our personal addition algorithm is far more efficient, using at the same time special properties of arithmetic operations and a clever number representation. I presume that anybody who would write an exact and comprehensive specification of his personal number addition algorithm including all efficiency tricks, special cases, 'if's and 'but's, would produce a document of several pages length that would not help anybody else in adding numbers. On the other hand, the primitive recursive definition of add is easy to understand and anybody can immediately realize that its interpretation as an algorithm is correct (although inefficient).

The advantage of a constructive, algorithmic specification is that it allows *rapid prototyping* in a straightforward way: if we want at the earliest possible phase of system development have some rudimentary version of the system running (e.g., in order to see if we have really specified what we wanted), then it is easy to translate this kind of specifications into any one imperative programming language. In fact, this can even be done automatically by some kind of '*specification compiler*'. Our INTAS system [41, 55] translates for instance SRDL specifications into PASCAL. The performance even of the inefficient translated SRDL specifications is not too bad compared with the performance of interpreters for sets of equations trying to deterministically simulate nondeterministic computations in systems of equations. We refuse to call this specification compilation an 'automatic implementation' since we feel that there is more in implementation than just simulating the specification. In fact, this would blur the borderline between specifying and programming, which is difficult enough to draw anyway.

10. Related work

An early predecessor of algebraic specification techniques is Burstall and Landin's paper [11]. Starting with the observation that many functions in computer applications can be described by *extending* a mapping on 'primitive objects' to a mapping

on 'data structures' they propose to consider data structures as algebras and programs as homomorphisms. The unique extension lemma (Lemma 1.1) is the key to a finite description of programs as homomorphisms provided that the underlying algebra is finitely generated (cf. the remarks on 'lifted operations' in Example 8.2). It is clear that not every data structure corresponds to a free algebra, so there may be mappings on generating sets which cannot be extended to homomorphisms. Burstall and Landin define algorithmically a partial functional *Extend* which computes the homomorphic extension if it exists. For this process, well-founded decompositions are introduced, which were the main motivation for the present work.

Later on, the concepts of *information hiding* or *data abstraction* were proposed as a basis of specification techniques. According to Liskov and Zilles [45], "an *abstract data type* defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type".

At this point, some fundamental questions arise, e.g.:

(1) What is an *abstract data type* mathematically? What is a *data type*?

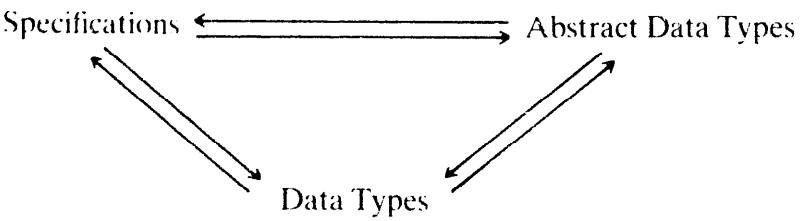
(2) What is a *specification* for an abstract data types?

(3) How to define the *semantics* of a specification, i.e., the abstract data type specified by it?

(4) What should be considered as an *implementation* of an abstract data type or of a specification?

(5) When is a specification *correct*? (and with respect to what?)

This means that we must define the objects and arrows in the (informal) diagram



After having fixed a *specification technique* by answering (1)–(4), some further questions come up naturally:

(6) What is the *power of the technique*, i.e., what are the (abstract) data types specifiable by it?

(7) How can *exceptions* be handled using the technique?

(8) How can specifications be structured, e.g., using *enrichments*, *extensions*, *parameterizations*? Under what circumstances are these structuring techniques 'safe' (i.e., do not change the semantics of an already existing specification)?

Of course, the quotation from [45] cited above seems to call for an algebraic (or at least model-theoretic) treatment of abstract data types. The algebraic approach to abstract data specification was first explored by the ADJ group [28], Guttag [30] and Zilles [63]. The similarity in these first approaches is that they all use equations

between terms over a signature with variables for the specification of operations. Apart from this, the approaches are quite different. Guttag and Zilles assume that a data type specification is based on some primitive data types which come from outside the specification and cannot be changed by it. In [28] there are also some kind of 'don't care''s in the specifications but they are rather interpreted as *parameters* which besides can be changed by the specifications (see the stack example, e.g.).

The ADJ group, having just accomplished some research on initial algebra semantics [27], defined a data type to be an algebra, an abstract data type the initial algebra in some class of algebras, a specification an equational specification **spec** = $\langle S, \Sigma, E \rangle$, and the semantics of **spec** to be the initial algebra $T_{\mathbf{spec}}$ in the algebraic variety $Alg_{\mathbf{spec}}$. The implementation and correctness concepts in this first paper are very primitive: an algebra A is an implementation of **spec** or $T_{\mathbf{spec}}$ iff it is isomorphic to $T_{\mathbf{spec}}$; **spec** is correct w.r.t. an algebra B iff B is isomorphic to $T_{\mathbf{spec}}$.

It turned out quickly that this implementation concept was unsatisfactory and that the simple-minded treatment of exceptions ('errors') in this paper was not correct. [29] introduced a better (but still insufficient) implementation concept and proposed to treat exceptions using so-called 'conditional error axioms'. These are not truly conditional axioms but use a special conditional which is itself a defined operation and which distinguishes 'error terms' and 'OK terms'. The application of this technique leads to an unbearable amount of equations for definition of an OK-predicate and for describing error propagation and OK propagation. Error recovery is not possible; this also applies to [23] where the (from the intuitive point of view) redundant equations of [29] have been avoided by the introduction of 'error algebras'. A satisfactory treatment of exceptions allowing error recovery was presented in [24].

Since the present paper does not deal with implementations, we do not further comment on this but simply refer to [17] where an appropriate extension to the ADJ framework has been discussed.

A very important concept which was introduced by [29] is the *canonical term* algebra. We have pointed out in this paper the similarities between canonical term algebras and decomposition algebras. At this point, we want to comment on the different motivation for the introduction of either kind of algebras: decomposition algebras were introduced in order to explicitly define recursive operations in algebras. Canonical term algebras were introduced as a tool for doing inductive proofs in abstract data types. It will be noted that usually the terms in a canonical term algebra C isomorphic to $T_{\mathbf{spec}}$ for **spec** = $\langle$**sig**, $E\rangle$ will not contain all of the operation symbols in **sig**. Those operation symbols appearing in C could be called *constructors* just in the sense of the **sig**-d-algebra because they generate the whole algebra C (or $T_{\mathbf{spec}}$). Unfortunately, there will be different canonical term algebras isomorphic to $T_{\mathbf{spec}}$ with different constructors, so we cannot speak of '*the*' constructors of **spec**. Constructors are a secondary concept in this approach whereas they are a primary one in the decomposition algebra approach. Canonical term algebras have been frequently used and investigated in the literature, sometimes without explicit

reference to their original appearance and sometimes with hints on the usefulness of selecting constructors a priori. We will only refer the reader to [13, 33, 53, 54, 57, 58, 61].

Concerning the power of the equational specification technique Majster [49] first claimed that an obviously simple data type could not be finitely specified using equations. This topic was further elaborated in [50]. In fact this argument is valid as long as we rule out so-called 'hidden functions', i.e., functions which belong to the signature (and hence to the algebra) but are not meant to belong to the data type being specified. This is, of course, in conflict to the informal equation 'data type = algebra' mentioned above. Furthermore, one may argue that extensive use of hidden functions tends to handicap the understandability of specifications [51]. With a limited use of hidden functions every computable data type can be finitely specified [3]. Frequently misquoted in this context (e.g., in [31]) is the definition of computable functions by Gödel and Herbrand using equations: while it is true that they employ the same syntax (i.e., equations), their semantics is very different from the semantics of equations in the algebraic specification literature: there is a specific deduction system associated with the equations, so the semantics is rather operational as opposed to factorizing by a congruence relation generated by the equations. Nevertheless, the Gödel–Herbrand result is frequently used in arguing that equations plus hidden functions can specify any computable operation. The somewhat unsafe feeling concerning the unlimited use of hidden functions still remains.

We claim to have tamed the hidden functions in our approach: first we draw a distinction between the data component and the function component of a specification, where the data component is hidden and the function component is exported. In the function definitions, hidden functions occur only as nested ('local') subdefinitions which is less dangerous and more intelligible than their ubiquitous presence in equational specifications.

Another problem with the initial algebra approach is the so-called 'junk'. Every new operation symbol in the signature introduces a countably infinite set of new terms into the term algebra, viz., all terms which contain this symbol. If we want to have the semantics of the specification isomorphic to some specific model of our data type (or software module)—see the definition of correctness cited above!—then we must take care to reduce all this 'junk' to other terms by special equations. This can require a considerable amount of equations, and it is clear that the danger of having contradictory sets of equations grows with the number of equations. On this background, it has been argued that it would be more profitable to have a *final (terminal) algebra semantics* [4, 9, 32, 37, 38, 62]. In order to have non-trivial final models, *primitive types* are introduced. These cannot be changed by a specification which defines only one new sort ('type of interest') together with operations over it and the primitive types. This kind of argument is also inherent to Guttag's [30, 31] and Zilles's work [63, 64] although they do not explicitly speak of terminal models. Again we claim that the 'junk problem' and its possible solution by terminal semantics

plays no role for our approach since here the defined operations do not belong to the signature and hence generate no junk.

Constructive approaches to abstract algebraic software specification have also been presented by Loeckx [46, 47], Mayr et al. [51] and Nourani [53]. (Here, we do not take into account some approaches where abstract objects are modelled in a fixed discipline, e.g., some kind of graphs.)

It was mentioned already in Section 2 that recursive definitions of operations are straightforward in term algebras but that unfortunately not all data structures correspond to a term algebra. If D is a data algebra of signature **sig** and h the unique homomorphism $h: T_{\mathbf{sig}} \to D$ then it will frequently be the case that $h(t_1) = h(t_2)$ for $t_1 \neq t_2$. Our approach has been to pass from $T_{\mathbf{sig}}$ to $T_{\mathbf{spec}}$ and to investigate methods for recursive definitions in algebras other than the term algebra. In contrast to this, Loeckx uses recursive definitions on the term algebra $T_{\mathbf{sig}}$. Every specification contains a so-called *acceptor function* defined recursively on the structure of $T_{\mathbf{sig}}$ and giving Boolean results. If we define $N \subseteq T_{\mathbf{sig}}$ as the set of terms for which the acceptor function yields true, then $h(t_1) = h(t_2)$ for $t_1, t_2 \in N$ implies $t_1 = t_2$ (in fact, the elements of N can be considered as some kind of normal forms). Since the question whether two terms denote the same data element is frequently important for the recursive specification of defined operations, every specification contains also an *equality predicate* which—like the acceptor function—is defined recursively on the structure of $T_{\mathbf{sig}}$. This is sort of doing the same work twice.

The 'operational replacement schemes' of [51] have great similarities to the structural recursive schemata presented here although Mayr and his colleagues seem to be hampered by the restriction that there can only be a single constructor ('Generator') for every sort. This frequently requires coding of several constructors into a single generator, which gives a somewhat artificial appearance to their specifications. In fact, some of them look rather like implementations than abstract specifications. Nourani [53] gives sufficient conditions for safe extensions of equational specifications. The key concept for this is the notion of *constructor signature* for canonical term algebras and the so-called extension by constructors which requires a non-ambiguous case distinction on constructor terms in order to define a new operation. However, our Example 4.2 might be a warning that the simulation of recursive definitions by sets of equations does not always have the expected semantics.

There is nothing in the literature to which our ideas on parameterizations could be compared, since all of the other publications study equational specifications. We briefly mention the work of Ehrich [15], Ganzinger [22], Hornung and Raulefs [32] and the ADJ group [19, 60].

Concerning our structural recursive definition language SRDL and the INTAS system [41, 55] which 'implements' it, we want to mention some alternative systems. First, there is Goguen's OBJ system [26] which has been successfully used in evaluating equational specifications. Term rewriting systems based on equational specifications have also been implemented in Berlin [48], Bonn [20] and Dortmund [56]. The

specification language CLEAR [10] by Burstall and Goguen involves concepts which are some orders of magnitude higher than those presented here; but this is only a first step towards the even more ambitious CAT project [25]. The AFFIRM system [52] should also be mentioned here, as well as the DAISTS system [21] where concrete models of abstract data types are matched against abstract equational specifications. It is a pleasure to acknowledge that the design goals of the experimental applicative language HOPE [12] have confirmed and influenced our decisions in the choice of language constructs for SRDL.

Acknowledgment

I want to thank H.-J. Kreowski for patiently and carefully reading an earlier version of this paper and for making numerous helpful suggestions. Also, I am very thankful to my colleague H. Petzsch for many discussions on the pragmatic aspects of the structural recursive specification method and to the referees for their suggestions. Thanks are also due to Sigrid Horenbeek for competently preparing the typescript.

References

For a complete bibliography, see [43]

[1] J.A. Bergstra, M. Broy, J.V. Tucker and M. Wirsing, On the power of algebraic specifications, in: *10th MFCS*, Lecture Notes in Comput. Sci. **118** (Springer, Berlin, 1981) pp. 193–204.

[2] J.A. Bergstra and J.J.C. Meyer, On specifying sets of integers, Rept. 80–21, Rijksuniv. Leiden, 1980.

[3] J.A. Bergstra and J.V. Tucker, On the adequacy of finite equational methods for data type specification, *SIGPLAN Notices* **14** (11) (1979) 13–18.

[4] J.A. Bergstra and J.V. Tucker, Initial and final algebra semantics for data type specification: Two characterization theorems, Rept. IW142\80, Math. Centr., Amsterdam, 1980.

[5] A. Bertoni, G. Mauri and P.A. Miglioli, Model-theoretic aspects of abstract data specification *26th. Coll. on Math. Logic in Programming*, Salgotarjan, 1978.

[6] A. Bertoni, G. Mauri and P.A. Miglioli, A characterization of abstract data as model-theoretic invariants, *6th ICALP*, Lecture Notes in Comput. Sci. **71** (Springer, Berlin, 1979) pp. 26–37.

[7] A. Bertoni, G. Mauri and P.A. Miglioli, Towards a theory of abstract data types: A discussion on problems and tools, *4e Coll. Internat. sur la Programmation*, Lecture Notes in Comput. Sci. **83** (Springer, Berlin, 1980) pp. 44–58.

[8] M. Broy, W. Dosch, H. Partsch, P. Pepper and M. Wirsing, Existential quantifiers in abstract data types, *6th ICALP*, Lecture Notes in Comput. Sci. **71** (Springer, Berlin, 1979) pp. 73–87.

[9] M. Broy and M. Wirsing, Initial versus terminal algebra semantics for partially defined abstract data types, Rept. TUM-18018, TU München, 1980.

[10] R.M. Burstall and J.A. Goguen, Semantics of CLEAR, a specification language, in: D. Bjorner, ed., *Abstract Software Specifications, Proc. Copenhagen Winter School*, Lecture Notes in Comput. Sci. **86** (Springer, Berlin, 1980) pp. 292–332.

[11] R.M. Burstall and P. Landin, Programs and their proofs—an algebraic approach, *Machine Intelligence* **4** (1969) 17–43.

[12] R.M. Burstall, D.B. MacQueen and D.T. Sannella, HOPE: an experimental applicative language Int. Rept. CSR-62-80, Dept. of Comput. Sci., Univ. of Edinburgh, 1980.

[13] C. Choppy, P. Lescanne and J.L. Remy, Improving abstract data types by appropriate choice of constructors, in: A. Bierman, C. Guiho and Y. Kodratoff, eds., *Automatic Program Construction Techniques* (MacMillan, New York, 1980).

[14] H.D. Ehrich, Extensions and implementations of abstract data type specifications, *7th MFCS*, Lecture Notes in Comput. Sci. **64** (Springer, Berlin, 1978) pp. 155–164.

[15] H.D. Ehrich, On the theory of specification, implementation and parameterization of abstract data types, *J. ACM* **29** (1982) 206–227.

[16] H.D. Ehrich, On realization and implementation, *10th MFCS*, Lecture Notes in Comput. Sci. **118** (Springer, Berlin, 1981) pp. 271–280.

[17] H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz, Algebraic implementation of abstract data types, *Theoret. Comput. Sci.* **20** (1982) 209–263.

[18] H. Ehrig, H.-J. Kreowski and P. Padawitz, Stepwise specification and implementation of abstract data types, *5th ICALP*, Lecture Notes in Comput. Sci. **62** (Springer, Berlin, 1978) pp. 205–226.

[19] H. Ehrig, H.-J. Kreowski, J.W. Thatcher, E.G. Wagner and J.B. Wright, Parameterized data types in algebraic specification languages, *7th ICALP*, Lecture Notes in Comput. Sci. **85** (Springer, Berlin, 1980) pp. 157–168.

[20] G. Eigemeier, Ch. Knabe, P. Raulefs and K. Tramer, Automatic implementation of algebraic specifications of abstract data types, MEMO SEKI-BN-79-11, Universität Bonn, 1979.

[21] J. Gannon, P. McMullin and R. Hamlet, Data abstraction implementation, specification, and testing, *TOPLAS* **3** (1981) 211–223.

[22] H. Ganzinger, Parameterized specifications: parameter passing and optimizing implementation, Rept. TUM-I8110, TU München, 1981.

[23] J.A. Goguen, Abstract errors for abstract data types, UCLA Semantics and Theory of Comput. Rept. 6, 1977; *Proc. IFIP Working Conf. on Formal Description of Programming Languages Concepts*, 1977.

[24] J.A. Goguen, Order sorted algebras: Exceptions and error sorts, coercions and overloaded operators, UCLA Semantics and Theory of Comput. Rept. 14, 1978.

[25] J.A. Goguen and R.M. Burstall, CAT, a system for the structured elaboration of correct programs from structured specifications, Techn. Rept. CSL-118, SRI International, 1980.

[26] J.A. Goguen and J.J. Tardo, An introduction to OBJ: A language for writing and testing formal algebraic program specifications, in: *Proc. IEEE Conf. Spec. for Reliable Software* (IEEE, 1979) pp. 170–189.

[27] J.A. Goguen, J.A. Thatcher, E.G. Wagner and J.B. Wright, Initial algebra semantics and continuous algebras, *J. ACM* **24** (1977) 68–95.

[28] J.A. Goguen, J.W. Thatcher, E.G. Wagner and J.B. Wright, Abstract data types as initial algebras and the correctness of data representations, *Proc. of Conf. on Computer Graphics, Pattern Recognition and Data Structures*, 1975.

[29] J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in: R. Yeh, ed., *Current Trends in Programming Methodology IV: Data Structuring* (Prentice-Hall, Englewood Cliffs, NJ, 1978) pp. 80–144.

[30] J.V. Guttag, The specification and application to programming of abstract data types, Ph.D. Thesis, University of Toronto, 1975.

[31] J.V. Guttag and J.J. Horning, The algebraic specification of abstract data types, *Acta Inform.* **10** (1978) 27–52.

[32] G. Hornung and P. Raulefs, Initial and terminal algebra semantics of parameterized abstract data type specifications with inequalities, *6ème Coll. Les Arbres en Algebre et Programmation*, Genova, 1981.

[33] G. Huet and J.M. Hullot, Proofs by induction in equational theories with constructors, *21st FOCS* (1980) 96–107.

[34] G. Huet and D.C. Oppen, Equations and rewrite rules: A survey, in: R. Book, ed., *Formal Languages: Perspectives and Open Problems* (Academic Press, New York, 1980).

[35] U.L. Hupbach, Rekursive Funktionen in mehrsortigen Peano-Algebren, *Elektron. Informationsverarb. Kybernet.* **14** (1978) 491–506.

[36] S. Kamin, Some definitions for algebraic data type specifications, *SIGPLAN Notices* **14** (3) (1979) 28–37.

[37] S. Kamin, Final data specification: a new data specification method, *7th POPL* (1980) 131–138.

[38] S. Kamin, Final data types and their specification, *TOPLAS* **5** (1983) 97–123.
[39] H.A. Klaeren, Datenräume mit algebraischer Struktur, Schriften zur Ang. Math. und Inform. 43, RWTH Aachen, 1978.
[40] H.A. Klaeren, Eine Klasse von Algebren mit struktureller Rekursion und ihre Anwendung bei der abstrakten Software-Spezifikation, Dissertation, RWTH Aachen, 1980.
[41] H.A. Klaeren, The SRDL specification experiment, *Workshop on Program Specification*, Lecture Notes in Comput. Sci. **134** (Springer, Berlin, 1981) pp. 282–293.
[42] H.A. Klaeren, On parameterized abstract software modules using inductively specified operations, *Coll. AFCET Les Math. de l'Informatique*, Paris (1982) 289–300.
[43] H.A. Klaeren, *Algebraische Spezifikation—Eine Einführung*, Springer Lehrbuch Informatik (Springer, Berlin, 1983).
[44] H.A. Klaeren and M. Schulz, Computable algebras, word problems, and canonical term algebras, *5. GI Fachtagung Theoretische Informatik*, Lecture Notes in Comput. Sci. **104** (Springer, Berlin, 1981) pp. 203–213.
[45] B. Liskov and S.M. Zilles, Programming with abstract data types, *SIGPLAN Notices* **9** (4) (1974) 50–59.
[46] J. Loeckx, Algorithmic specifications of abstract data types, *8th ICALP*, Lecture Notes in Comput. Sci. **115** (Springer, Berlin, 1981) pp. 129–147.
[47] J. Loeckx, Implementations of abstract data types and their verification, *11. GI-Jahrestagung*, IFB **50** (1981) 96–108.
[48] M. Loewe, M. Reisin and K.P. Hasler, Algebraic specification of a user-controlled interpreter for algebraic specifications, Report, TU Berlin, 1982.
[49] M.E. Majster, Limits of the 'algebraic' specification of abstract data types, *SIGPLAN Notices* **12** (10) (1977) 37–42.
[50] M.E. Majster, Data types, abstract data types and their specification problem, *Theoret. Comput. Sci.* **8** (1979) 89–127.
[51] H.C. Mayr, P.C. Lockemann and K.R. Dittrich, Operational replacement schemes—a practice-oriented approach to the specification of abstract data types, Bericht Nr. 11/80, Fakultät Informatik, Universität Karlsruhe, 1980.
[52] D.R. Musser, Abstract data type specification in the AFFIRM system, *IEEE Trans. Software Engrg* **SE-6** (1) (1980).
[53] F. Nourani, Constructive extension and implementation of abstract data types and algorithms, Ph.D. Thesis, Techn. Rept. UCLA-ENG-7945, Univ. of California, Los Angeles, 1979.
[54] P. Padawitz, Proving the correctness of implementations by exclusive use of term algebras, Bericht Nr. 79-8, Fachbereich Informatik, TU Berlin, 1979.
[55] H. Petzsch, INTAS—Ein System zur Interpretation algebraischer Spezifikationen, *Berichte des Lehrstuhls für Informatik II*, **5** (RWTH Aachen, 1981).
[56] U. Pletat, G. Engels and H.-D. Ehrich, Operational semantics of algebraic specifications with conditional equations, Bericht Nr. 118/81, Abteilung Informatik, Universität Dortmund, 1981.
[57] J.L. Remy and P.A.S. Veloso, An economical method for comparing data type specifications, *SIGPLAN Notices* **16** (5) (1981) 39–42.
[58] J.L. Remy and P.A.S. Veloso, Comparing abstract data type specifications via their normal forms, *J. Comput. Information Sci.* **11** (1982) 141–154.
[59] M. Schulz, Berechenbare Algebren, Wortprobleme und kanonische Termalgebren, Diplomarbeit, RWTH Aachen, 1981.
[60] J.W. Thatcher, E.G. Wagner and J.B. Wright, Data type specification: Parameterization and the power of specification techniques, *10th STOC* (1978) 119–132.
[61] P.A. Veloso and T.H. Pequeno, Don't write more axioms than you have to: A methodology for the complete and correct specification of abstract data types, with examples, *Monografias em Ciencia da Computacao* **10** (1979); Extended Abstract in: *Proc. Internat. Comp. Symp.*, Nankang (China) (1978).
[62] M. Wand, Final algebra semantics and data type extensions, *J. Comput. System Sci.* **8** (1979) 27–44.
[63] S.N. Zilles, Algebraic specification of data types, *Project MAC Progress Rept.* **11** (MIT, 1974) 28–52.
[64] S.N. Zilles, An introduction to data algebras, in: D. Bjorner, ed., *Abstract Software Specifications*, Lecture Notes in Comput. Sci. **86** (Springer, Berlin, 1980) pp. 248–272.